

# C.L.Tondo \* S.E. Gimpel

Műszaki Könyvkiadó







*Clovis L. Tondo – Scott E. Gimpel*

# **C programozási gyakorlatok**

---

**Megoldások**

---

**Brian W. Kernighan és Dennis M. Ritchie**

---

**A C programozási nyelv című könyvének**

---

**feladataihoz**

---

Műszaki Könyvkiadó, Budapest, 1988



Az eredeti mű:

Clovis L. Tondo—Scott E. Gimpel:

The C ANSWER BOOK, Solutions to the Exercises in The C Programming Language  
by Brian W. Kernighan and Dennis M. Ritchie

Original English language edition published by Prentice—Hall, Inc.

Copyright © 1985 by Prentice—Hall, Inc.

All Rights Reserved

Lektorálta: Dr. Körösi István okl. villamosmérnök

© Hungarian translation Siegler András, 1988

ETO: 519.682 C (076)

ISBN: 963 10 7777 2



# Tartalom

---

|             |                                                |     |
|-------------|------------------------------------------------|-----|
|             | Előszó .....                                   | 7   |
| 1. fejezet: | Alapismeretek .....                            | 9   |
| 2. fejezet: | Típusok, operátorok és kifejezések .....       | 37  |
| 3. fejezet: | Vezérlési szerkezetek .....                    | 47  |
| 4. fejezet: | Függvények és programstruktúra .....           | 55  |
| 5. fejezet: | Mutatók és tömbök .....                        | 68  |
| 6. fejezet: | Struktúrák .....                               | 100 |
| 7. fejezet: | Bevitel és kivitel .....                       | 114 |
| 8. fejezet: | Csatlakozás a UNIX operációs rendszerhez ..... | 123 |
|             | Tárgymutató .....                              | 137 |

Ez a könyv feladatok MEGOLDASAIT tartalmazza. Megoldásokat mutat be Brian W. Kernighan és Dennis M. Ritchie A C programozási nyelv című könyvének (Műszaki Könyvkiadó, 1985) példafeladataihoz. (A könyvre a továbbiakban K-R néven hivatkozunk.) A C programozási gyakorlatok figyelmes tanulmányozása – együtt a K-R könyvvel – elvezeti az Olvasót a C programozási nyelv megértéséhez és jól használható programozási fogásokra tanít.

Tanulási módszerként azt ajánljuk az Olvasónak, hogy a K-R könyvből tanulja a C nyelvet, dolgozza ki az ott közölt gyakorlatokat, majd tanulmányozza a jelen könyvben közölt megoldásokat. A megoldások kidolgozása során ügyeltünk arra, hogy csak olyan C nyelvi elemeket használjunk fel, amelyek a feladat K-R könyvbéli megjelenésének helyén már ismertek. Célunk az volt, hogy a K-R könyvvel együtt haladjunk. A későbbiekben, amikor az Olvasó már többet tud a nyelvről, bizonyosan a közöltek-nél jobb megoldásokat is fog majd találni. Például mindaddig, amíg az

```
if (kifejezés)
    1._utasítás
else
    2._utasítás
```

tipusú szerkezet a K-R könyv 33. oldalán elő nem fordul, ezt a szerkezetet nem használjuk. A fenti struktúra használatával természetesen jobb megoldást adhatunk az 1.6., 1.7. és 1.8. gyakorlatokra (K-R 31. oldal). Egyes esetekben azonban e kényszerből eltekintünk.

A K-R könyv 28. oldalán találjuk a szimbolikus állandók leírását. Amikor egy megoldásban olyan szabványos szimbolikus állandó fordul elő, mint például az EOF, mindig megjelenik a

```
#include <stdio.h>
```

sor is. Az `stdio.h` elnevezésű állomány (header file) a szabványos be-kiviteli (I/O) könyvtár által használt makrókat és szimbolikus állandókat tartalmazza. Az `stdio.h` állományra vonatkozó ismeretek a K-R könyv 150. és 171. oldalán találhatók.



A megoldásokhoz magyarázat is tartozik. Feltételezzük, hogy az Olvasó már elsajátította a gyakorlatokat megelőző ismeretanyagot. Törekedtünk arra, hogy ne ismételjük meg azt, ami a K-R könyvben található, hanem az egyes megoldások legfontosabb elemeire mutassunk rá.

Nem lehet egy programozási nyelvet a nyelv elemeinek pusztá elolvasásával megtanulni. Ehhez programozni is kell - az Olvasónak kell elkészítenie saját programját és tanulmányoznia kell másokét is. A nyelv legjobb tulajdonságainak bemutatását tartottuk szem előtt, modularizált programokat készítettünk, kiterjedten használtuk a könyvtári rutinokat és a programjainknak olyan megjelenést kölcsönöztünk, hogy az világossá tegye a logikai felépítést.

Hálával tartozunk a C nyelv úttörőinek, Brian W. Kernighannek és Dennis M. Ritchienek, akiknek gondolatmenete és stílusa munkánk során irányadó volt. Ugyancsak köszönjük Brian W. Kernighannek, hogy munkánk eredményét gondosan ellenőrizte. Köszönetünket fejezzük ki továbbá szerkesztőnknek, Karl V. Karlstromnek.

Köszönetet érdemel barátunk, Ramsey W. Haddad a Stanford Egyetemről, mert a reméltnél sokkal mélyebben foglalkozott könyvünk kéziratával és olyan problémákat fedezett fel, amelyek korábban elkerülték figyelmünket. Mindig nagyra értékeljük baráti támogatását és együttműködését.

Végül köszönetünk illeti Phillip M. Adamst, Patricia A. Gatest, Stephen L. Mackeyt és szüleinket, akik munkánk során különböző formában segítettek bennünket.

Clovis L. Tondo

Scott E. Gimpel

## 1. FEJEZET

# Alapismeretek

---

### 1.1. Gyakorlat: (K-R 20. oldal)

Futtassa le a programot a saját rendszerén! Kisérletezzzen a program egyes részeinek elhagyásával, hogy meglássa, milyen hibaüzenetek érkeznek.

```
main()
{
    printf("figyelem, emberek");
}
```

A fenti példában az újsor (\n) karakter hiányzik. Ennek következtében a kimenetet nem követi soremelés (újsor).

```
main()
{
    printf("figyelem, emberek\n")
}
```

A második példában a printf() utáni pontosvessző hiányzik. Minden C nyelvi utasítást pontosvesszőnek kell lezárnia (K-R 23. oldal). Ennek hiányát a fordítóprogramnak kell észlelnie és a megfelelő hibaüzenettel kell reagálnia.

```
main()
{
    printf("figyelem, emberek\n");
}
```

A harmadik példában a \n-et követő idézőjel (") hiányzik. Ennek következtében az aposztróf jel, a záró kerek zárójel és a pontosvessző a karakterlánc részének tekintődik. A fordítónak ezt hibaként kell észlelnie és hibaüzenetet kell küldenie arról, hogy az idézőjel hiányzik, vagy arról, hogy a karakterlánc túl hosszú.

### 1.2. Gyakorlat: (K-R 22. oldal)

Próbálja ki, mi történik, ha a printf argumentum-karakterlánc \x-et tartalmaz, ahol x a fenti listában nem szereplő karakter!



```

main()
{
    printf("figyelem, emberek\a");
    printf("figyelem, emberek\7");
    printf("figyelem, emberek\?");
}

```

A C referencia-kézikönyv szerint (A. függelék, K-R 186. oldal)

ha a fordított törtvonalat (backslash karaktert) követő karakter nem a megadottak közül való, a fordított törtvonal nem vevődik figyelembe.

A C programozási nyelv - Referencia-kézikönyv (írta Dennis Ritchie) 1983 júniusi kiadása szerint (ez bizonyos változtatásokkal azonos a K-R könyv A. függelékével)

ha a törtvonalat követő karakter nem a megadottak közül való, a program működése meghatározatlan.

E kísérlet eredménye függ a fordítóprogramtól. Egy lehetséges eredmény:

```

figyelem, emberekfigyelem, emberek<BELL>figyelem,
emberek?

```

ahol <BELL> az ASCII 7 által előidézett rövid hangjelzés.

(A fenti és a további programokban a megjegyzéseket ékezet használata nélkül szerepeltetjük. Ezáltal a programok az itt leírtakkal karakterre azonosan valószínűsíthetők meg az Olvasó számítógépén. - A fordító megjegyzése.)

### 1.3. Gyakorlat: (K-R 25. oldal)

Módosítsuk úgy a hőmérséklet-átszámító programot, hogy az a táblázat fölé fejléceket is nyomtasson!

```

main()      /* Fahrenheit-Celsius tablázat fahr = 0, 20,
                                     ..., 300 értékekre */
{
    int lower, upper, step;
    float fahr, celsius;

    lower = 0;          /* a hőmérséklet tablázat alsó
                                     határa */
    upper = 300;        /* felső határ */
    step = 20;          /* lépésméret */

    printf("Fahr Celsius\n");
}

```

```

fahr = lower;
while (fahr <= upper) {
    celsius = (5.0/9.0) * (fahr-32.0);
    printf("%4.0f %6.1f\n", fahr, celsius);
    fahr = fahr + step;
}
}

```

Ha a ciklusutasítás elé a

```
printf("Fahr Celsius\n");
```

sort iktatjuk be, a táblázat egyes oszlopai fölé fejléc íródik. A program további része azonos a K-R könyv 22. oldalán közölttel.

#### 1.4. Gyakorlat: (K-R 25. oldal)

Írjunk programot a korábbi példának megfelelő Celsius-Fahrenheit táblázat kinyomtatására!

```

main()          /* Celsius-Fahrenheit tablázat celsius = 0,
                  20,..., 300 értékekre */
{
    int lower, upper, step;
    float fahr, celsius;

    lower = 0;          /* a hőmérséklet tablázat alsó
                           határa */
    upper = 300;        /* felső határ */
    step = 20;          /* lépésméret */

    printf("Celsius Fahr\n");
    celsius = lower;
    while (celsius <= upper) {
        fahr = (9.0*celsius) / 5.0 + 32.0;
        printf("%4.0f %6.1f\n", celsius, fahr);
        celsius = celsius + step;
    }
}

```

A program olyan táblázatot készít, amely a Celsius fokban mért hőmérsékleteket (0 ... 300 között) és Fahrenheit értékekben kifejezett megfelelőiket tartalmazza. Az ismert Celsius értékekből kiindulva a Fahrenheit értékek kiszámítása a

```
fahr = (9.0*celsius) / 5.0 + 32.0;
```

utasítással történik. A megoldás ugyanazt a logikai felépítést követi, mint a Fahrenheit-Celsius táblázatot kinyomtató program (K-R 22. oldal). A lower, upper és step egész típusú változók a celsius változó alsó és felső korlátját valamint lépésméretét jelentik. A celsius nevű változó kezdeti értéke, az alsó korlát és az



ennek megfelelő Fahrenheit hőmérséklet a while cikluson belül számítható ki. A Celsius és a megfelelő Fahrenheit érték kiíródik és a celsius változó a lépésnagysággal megnövekszik. A while ciklus mindaddig ismétlődik, amíg a celsius változó meg nem haladja a felső korlátot.

### 1.5. Gyakorlat: (K-R 26. oldal)

Módosítsuk úgy a hőmérséklet-átszámító programot, hogy az a táblázatot fordított sorrendben, tehát 300 foktól 0 fokig nyomtassa ki.

```
main()          /* Fahrenheit-Celsius tablázat fahr = 300,
                  280,..., 0 értékekre */
{
    int fahr;

    for (fahr = 300; fahr >= 0; fahr = fahr - 20)
        printf("%4d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Az egyetlen módosítás a programban:

```
for (fahr = 300; fahr >= 0; fahr = fahr - 20)
```

Az utasítás első része, vagyis

```
fahr = 300
```

a felső korlátot állítja be a Fahrenheit (fahr) változó kezdeti értékéül. A második rész, vagyis a ciklust vezérlő feltétel

```
fahr >= 0
```

ellenőrzi, hogy a Fahrenheit változó elérte vagy átlépte-e az alsó korlátját. A for ciklus mindaddig ismétlődik, amíg a feltétel fennáll. Az újrainicializáló lépés, azaz

```
fahr = fahr - 20
```

a lépésnagysággal csökkenti a Fahrenheit változót.

### 1.6. Gyakorlat: (K-R 31. oldal)

Írjunk olyan programot, amely megszámolja a szóközöket, a tab- és az újsor karaktereket!

```
#include <stdio.h>
```

```
main()          /* szokozok, tab-ok es ujsorok szamlalasa */
{
    int c, nb, nt, nl;
```

```

nb = 0; /* szokozok szama */
nt = 0; /* tab-ok szama */
nl = 0; /* ujsorok szama */
while ((c = getchar()) != EOF) {
    if (c == ' ')
        ++nb;
    if (c == '\t')
        ++nt;
    if (c == '\n')
        ++nl;
}
printf("%d %d %d\n", nb, nt, nl);
}

```

A program az nb, nt és nl egész típusú változók segítségével számlálja a szóközöket, a tab- illetve újsor-karaktereket. A program elején a három automatikus változó kezdeti értékekül 0-t adunk meg.

A while ciklus törzsében minden egyes, a bemenetről érkező szóköz, tab- és újsor-karakter előfordulása feljegyződik. Minden alkalommal az összes ciklusbeli if utasítás végrehajtódik. Ha a beolvasott karakter nem szóköz, nem tab- és nem is újsor-karakter, akkor a program nem csinál semmit. Ha viszont a bejövő karakter a felsorolt három közül való, akkor a megfelelő számláló inkrementálódik. Az állomány vége (EOF) karakter észlelésének hatására az eredmények kiíródnak.

Az if-else utasítást a K-R könyv csak a 33. oldalon ismerteti. Ennek ismeretében a megoldás az alábbi lehet:

```

#include <stdio.h>

main() /* szokozok, tab-ok es ujsorok szamlalasa */
{
    int c, nb, nt, nl;

    nb = 0; /* szokozok szama */
    nt = 0; /* tab-ok szama */
    nl = 0; /* ujsorok szama */
    while ((c = getchar()) != EOF) {
        if (c == ' ')
            ++nb;
        else if (c == '\t')
            ++nt;
        else if (c == '\n')
            ++nl;
    }
    printf("%d %d %d\n", nb, nt, nl);
}

```



## 1.7. Gyakorlat: (K-R 31. oldal)

Írjunk olyan programot, amely a bemenetet átmásolja a kimenetre, miközben az egy vagy több szóközből álló karakterláncokat egyetlen szóközzel helyettesíti!

```
#include <stdio.h>
#define NONBLANK 'a'

main() /* szokozsorozat helyettesítése egyetlen szóközzel */
{
    int c, lastc;

    lastc = NONBLANK;
    while ((c = getchar()) != EOF) {
        if (c != ' ')
            putchar(c);
        if (c == ' ')
            if (lastc != ' ')
                putchar(c);
        lastc = c;
    }
}
```

A program a szóközöket keresi és amennyiben talál egyet, ellenőrzi, hogy nem követi-e közvetlenül egy másik szóköz. Ha igen, akkor a szóközt figyelmen kívül hagyja, egyébként kiírja. Ez az eljárás gondoskodik arról, hogy minden egyedülálló szóköz kiíródjék, csakúgy, mint a több szóközből álló láncok első tagjaiként előforduló szóközök midegyike.

A `c` nevű egész típusú változó tartalmazza a bemenetről legutoljára érkezett karakter ASCII értékét, míg a `lastc` változóban mindig az előzőnek beérkezett karakter ASCII értéke van. A `NONBLANK` szimbolikus állandó segítségével a `lastc` változó kezdeti értéke egy tetszőleges nem szóköz karakter ASCII értéke lesz (a fenti programban az `'a'`-t használtuk).

A `while` ciklus törzsében előforduló első `if` utasítás a nem szóköz karakterekkel foglalkozik. Ezeket a program kiírja. A második `if` feladata a szóközök kezelése, míg a harmadik `if` utasítás azt vizsgálja, hogy egyedülálló szóközről, illetve egy szóközlánc első tagjáról van-e szó. Végezetül `lastc` megkapja az új értékét, és az eljárás megismétlődik.

Az `if-else` utasítást a K-R könyv a 33. oldalon mutatja be. Ennek ismeretében a megoldás a következő lehet:

```

#include <stdio.h>
#define NONBLANK 'a'

main() /* szokozsorozat helyettesítése egyetlen
                                             szokozsal */
{
    int c, lastc;

    lastc = NONBLANK;
    while ((c = getchar()) != EOF) {
        if (c != ' ')
            putchar(c);
        else if (lastc != ' ')
            putchar(c);
        lastc = c;
    }
}

```

A logikai VAGY (||) operátort a K-R könyv a 32. oldalon mutatja be. Ennek ismeretében a megoldás a következő lehet:

```

#include <stdio.h>
#define NONBLANK 'a'

main() /* szokozsorozat helyettesítése egyetlen
                                             szokozsal */
{
    int c, lastc;

    lastc = NONBLANK;
    while ((c = getchar()) != EOF) {
        if (c != ' ' || lastc != ' ')
            putchar(c);
        lastc = c;
    }
}

```

### 1.8. Gyakorlat: (K-R 31. oldal)

Írjunk olyan programot, amely minden egyes tab-karaktert a >, visszaléptetés (backspace), - háromkarakteres sorozattal helyettesít, ami áthúzott >-ként fog megjelenni, továbbá, amely a visszaléptetés-karaktereket a hasonlóan áthúzott < szimbólummal helyettesíti! Ezáltal a tab-karakterek és a visszaléptetések láthatóvá válnak.

```

#include <stdio.h>

main() /* tab-ok és visszaléptetések helyettesítése
                                             háromkarakteres sorozattal */
{
    int c;

```



```

while ((c = getchar()) != EOF) {
    if (c == '\t')
        printf("-\b>");
    if (c == '\b')
        printf("-\b<");
    if (c != '\b')
        if (c != '\t')
            putchar(c);
}
}

```

Három eset fordulhat elő, és ennek megfelelően három feltételt vizsgál meg a program a while ciklus belsejében. A bemenetről érkező karakter vagy tab, vagy visszaléptetés, vagy pedig bármi egyéb. Ha a program tab- vagy visszaléptetés-karakterre bukkan, a neki megfelelő háromkarakteres sorozatot írja ki. Ha a karakter sem tab, sem pedig visszaléptetés, úgy azt a kimenetre írja. A while ciklus az állomány vége jelzés észleléséig ismétlődik.

Az olyan képernyős megjelenítőkön, amelyeken a felülírás nem lehetséges, a `-\b>` karaktersorozat helyett egyetlen `>` jelenik meg, míg a `<\b-` helyett csupán `-`. Célszerűbb, ha a `-\b>` és `-\b<` karaktersorozatot használjuk, mivel így legalább a nyíl hegye megmarad.

Az if-else utasítást a K-R könyv a 33. oldalon mutatja be. Ennek ismeretében a megoldás a következő lehet:

```

#include <stdio.h>

main() /* tab-ok es visszaleptetese helyettesitese
        háromkarakteres sorozattal */
{
    int c;

    while ((c = getchar()) != EOF) {
        if (c == '\t')
            printf("-\b>");
        else if (c == '\b')
            printf("-\b<");
        else
            putchar(c);
    }
}

```

### 1.9. Gyakorlat: (K-R 33. oldal)

Hogyan ellenőrizhetjük a szavakat számláló programot? Milyen korlátai lehetségesek?

A szavakat számláló program ellenőrzését kezdjük úgy, hogy nem adunk meg bemenetet. Ekkor a kimeneten 0 0 0 -

nak kell megjelennie (nulla újsor, nulla szó, nulla karakter).

Ezután próbáljuk ki a programot egy egykarakteres szóval. Ekkor a kimenetnek 1 1 2 -nek kell lennie (egy újsor, egy szó, két karakter - tudniillik egy betű, amit egy újsor-karakter követ).

Próbálkozzunk egy kétkarakteres szóval. Ekkor a kimenetnek 1 1 3 -nak kell lennie (egy újsor, egy szó, három karakter - tudniillik két karakter, amit egy újsor karakter követ).

Végezetül próbáljunk ki két db egykarakteres szót (a kimenetnek 1 2 4 -nek kell lennie) valamint két db egykarakteres szót úgy is, hogy soronként egy karaktert adunk meg (a kimeneten 2 2 4 -nek kell megjelennie).

Néhány speciális eset:

- nincs bemenet
- nincs megadva egyetlen szó sem, csak újsorok
- nincs megadva egyetlen szó sem, csak szóközök, tab-karakterek és újsorok
- soronként egyetlen szó - nincsenek szóköz- és tab-karakterek
- a bemenet legelején kezdődő szó
- néhány szóköz után kezdődő szó

#### 1.10. Gyakorlat: (K-R 33. oldal)

Írjunk olyan programot, amely külön-külön sorokban nyomtatja ki a bemenetére érkező szavakat!

```
#include <stdio.h>
#define YES 1
#define NO 0

main() /* szavak kinyomtatása soronként */
{
    int c, inword;

    inword = NO;
    while ((c = getchar()) != EOF) {
        if (c == ' ' || c == '\n' || c == '\t') {
            if (inword == YES) {
                putchar('\n'); /* a szó vége */
                inword = NO;
            }
        } else if (inword == NO) {
            inword = YES; /* a szó kezdete */
            putchar(c);
        } else /* a szon belül van */
            putchar(c);
    }
}
```



Az inword egész típusú logikai változó, amely azt jelzi, hogy a program éppen egy szón belül van-e vagy sem. A program elején az inword kezdeti értéke NO, mivel nem tudjuk, hogy hol kezdődik szó.

Az első if utasítás, vagyis

```
if (c == ' ' || c == '\n' || c == '\t')
```

eldönti, hogy c szóelválasztó karakter-e. Ha az, akkor a második if utasítás, vagyis

```
if (inword == YES)
```

azt határozza meg, hogy a megtalált szóelválasztó egy szó végét jelenti-e. Ha igen, akkor egy újsor íródik ki és inword új értéket kap, egyébként pedig nem történik intézkedés.

Ha c nem szóelválasztó, akkor vagy egy szó első karaktere, vagypedig valamilyen másik karakter a szóban. Ha c egy új szó kezdete, akkor inword új értéket kap. A karakter minden esetben kiírásra kerül.

### 1.11. Gyakorlat: (K-R 33. oldal)

Módosítsuk a szavakat számláló programot úgy, hogy a "szó" egy jobb definícióját használjuk, például azt, hogy a szó betűk, számjegyek és aposztrófok jelekkel vagy szimbólumokkal kezdődő sorozata.

```
#include <stdio.h>
#define YES 1
#define NO 0

main() /* bejovo sorok, szavak, karakterek
                                             szamlalasa */
{ /* csak ASCII */
    int c, nl, nw, nc, inword;

    inword = NO;
    nl = nw = nc = 0;
    while ((c = getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c == '\t')
            inword = NO;
        else if (inword == NO) {
            if ((c >= 'a' && c <= 'z') || /* kisbetu? */
                (c >= 'A' && c <= 'Z')) { /* nagybetu? */
                inword = YES;
                ++nw;
            }
        }
        } else if ((c >= 'a' && c <= 'z') ||
```

```

/* kisbetu? */
(c >= 'A' && c <= 'Z') || /* nagybetu? */
(c >= '0' && c <= '9') || /* szamjegy? */
c == '\\' ) /* aposztróf? */
/* szon belül */
;
else
    inword = NO;
}
printf("%d %d %d\n", nl, nw, nc);
}

```

A szó javított definíciójának használata szükségessé teszi, hogy három további utasítást iktassunk a programba.

```

if ((c >= 'a' && c <= 'z') ||
    (c >= 'A' && c <= 'Z'))

```

azt határozza meg, hogy egy szó első karaktere nagybetű, illetve kisbetű-e (csak ASCII lehet). Ha ezek valamelyike, akkor egy új szó kezdődik: inword új értéket kap és a szószámláló eggyel növekszik.

Az

```

else if ((c >= 'a' && c <= 'z') ||
         (c >= 'A' && c <= 'Z') ||
         (c >= '0' && c <= '9') ||
         c == '\\')

```

ellenőrzés elvégzésére csak akkor kerül sor, ha a program éppen egy szó belsejét dolgozza fel és a szóköz, újsor-, illetve tab-karaktertől különböző karaktert talált. Ez a vizsgálat garantálja, hogy a szó betűk, számjegyek és aposztrófok (csak ASCII karakterek) sorozata.

Az utolsó új utasítás,

```

else
    inword = NO;

```

a szóközön, újsoron és tab-karakteren kívüli nemszó-karakterek előfordulása miatt szükséges. Ha e további karakterek valamelyikére akad a program, az inword új értéket kap, jelezve, hogy a program most kívülre került a szón.

## 1.12. Gyakorlat: (K-R 35. oldal)

Írjunk olyan programot, amely kinyomtatja a bemenetén előforduló szavak hosszúságának hisztogramját! A legegyszerűbb, ha a hisztogramot vízszintesen rajzoljuk – a függőleges irányú rajzolás nehezebb feladat.



```

#include <stdio.h>
#define MAXHIST 15 /* hisztogram max. hossza */
#define MAXWORD 11 /* szo max. hossza */
#define YES 1
#define NO 0

main() /* vizszintes hisztogram */
{
    int c, i, inword, nc;
    int len; /* oszlop hossza */
    int maxvalue; /* wl[] max. erteke */
    int wl[MAXWORD]; /* szohossz szamlalok */

    inword = NO;
    nc = 0; /* karakterszam egy szoban */
    for (i = 0; i < MAXWORD; i++)
        wl[i] = 0;
    while ((c = getchar()) != EOF) {
        if (c == ' ' || c == '\n' || c == '\t') {
            inword = NO;
            if (nc > 0 && nc < MAXWORD)
                ++wl[nc];
            nc = 0;
        } else if (inword == NO) {
            inword = YES;
            nc = 1; /* uj szo kezdete */
        } else
            ++nc; /* szon belül */
    }
    maxvalue = 0;
    for (i = 1; i < MAXWORD; i++)
        if (wl[i] > maxvalue)
            maxvalue = wl[i]; /* elofordult max. ertekek */
    for (i = 1; i < MAXWORD; i++) {
        printf("%5d - %5d : ", i, wl[i]);
        if (wl[i] > 0) {
            if ((len = (wl[i] * MAXHIST / maxvalue)) <= 0)
                len = 1;
        } else
            len = 0;
        while (len > 0) {
            putchar('*');
            --len;
        }
        putchar('\n');
    }
}

```

A szó végét szóköz-, újsor- vagy tab-karakter jelenti. Ha a program egy szót talál ( $nc > 0$ ) és a szó hossza kisebb, mint a maximális szóhosszúság ( $nc < MAXWORD$ ), akkor a megfelelő szóhossz-számláló eggyel növekszik ( $++wl[nc]$ ). Ha a program nem talált szót (csak szóköz(öke)t, újsor(oka)t vagy tab-karakter(eke)t, illetve a szó hossza meghaladja a megadott tartományt ( $nc \geq MAXWORD$ ), akkor a szóhossz-számláló értéke nem változik.

Ha a program minden szót beolvasott, az előfordult maxi-

mális érték (maxvalue) meghatározása a wl tömbből törté-  
nik.

A len végzi MAXHIST és maxvalue szerint az érték skálá-  
zását wl[i] -be. Ha wl[i] nagyobb, mint 0, akkor leg-  
alább egy csillagot nyomtat ki.

Ezek után a program kinyomtatja a vízszintes hisztogra-  
mot.

```
#include <stdio.h>
#define MAXHIST 15 /* hisztogram max. hossza */
#define MAXWORD 11 /* szo max hossza */
#define YES 1
#define NO 0

main() /* vertikális hisztogram */
{
    int c, i, inword, j, nc;
    int len; /* oszlop hossza */
    int maxvalue; /* wl[] max. erteke */
    int wl[MAXWORD]; /* szohossz-szamlalok */

    inword = NO;
    nc = 0; /* karakterszam egy szoban */
    for (i = 0; i < MAXWORD; i++)
        wl[i] = 0;
    while ((c = getchar()) != EOF) {
        if (c == ' ' || c == '\n' || c == '\t') {
            inword = NO;
            if (nc > 0 && nc < MAXWORD)
                ++wl[nc];
            nc = 0;
        } else if (inword == NO) {
            inword = YES;
            nc = 1; /* uj szo kezdete */
        } else
            ++nc; /* szon belül */
    }
    maxvalue = 0;
    for (i = 1; i < MAXWORD; i++)
        if (wl[i] > maxvalue)
            maxvalue = wl[i]; /* elofordult max. erteke */
    for (i = MAXHIST; i > 0; i--) {
        for (j = 1; j < MAXWORD; j++)
            if ((wl[j] * MAXHIST / maxvalue) >= i)
                printf(" * ");
            else
                printf(" ");
        putchar('\n');
    }
    for (i = 1; i < MAXWORD; i++)
        printf("%4d ", i);
    putchar('\n');
    for (i = 1; i < MAXWORD; i++)
        printf("%4d ", wl[i]);
    putchar('\n');
}
```



Ez a megoldás vertikális hisztogramot nyomtat ki. maxvalue meghatározásáig hasonló az előző programhoz. Ezek után a wl tömb minden elemét be kell skálázni és gondoskodni kell arról, hogy mindegyik elemnek megfelelően egy-egy csillag legyen kinyomtatva. Erre azért van szükség, mert az összes oszlop kinyomtatása egyidejűleg történik (vertikális hisztogram készül). A program két utolsó for ciklusa kiírja wl minden elemének az indexét és az értékét.

### 1.13. Gyakorlat: (K-R 37. oldal)

Írjunk olyan programot, amely a beolvasott szöveget kisbetűssé alakítja át egy lower(c) függvény segítségével. A program térjen vissza c-vel, ha c nem betű és c kisbetűs megfelelőjét adja vissza, ha c betű!

```
#include <stdio.h>

main() /* bemenet atalakitasa kisbetusse */
{
    int c;

    while ((c = getchar()) != EOF)
        putchar(lower(c));
}

lower(c)
int c;
{
    if (c >= 'A' && c <= 'Z')
        return(c - 'A' + 'a');
    else
        return(c);
}
```

Amennyiben c az 'A' - 'Z' tartományban van, akkor a lower függvény a c kisbetűs megfelelőjével tér vissza. A

$c - 'A' + 'a'$

kifejezés az ASCII karakterkészleten belül minden nagybetűs karaktert a neki megfelelő kisbetűssé alakít át. Ha c bármilyen más karakter, a függvény magát a c-t adja vissza.

A fő rutinban putchar a lower által visszaadott értéket írja ki.

A 2.10. gyakorlatban újra megírjuk a lower függvényt, mégpedig a feltételes kifejezés felhasználásával (K-R 59. oldal).

#### 1.14. Gyakorlat: (K-R 40. oldal)

Módosítsuk a leghosszabb sort kereső program fő rutinját oly módon, hogy helyesen írja ki tetszőlegesen hosszú bemeneti sorok hosszát és a szövegből annyit, amennyi csak lehetséges.

```
#include <stdio.h>
#define MAXLINE 1000

main() /* megkeresi a leghosszabb sort */
{
    int len; /* aktuális sorhossz */
    int max; /* eddigi max. hossz */
    char line[MAXLINE]; /* aktuális bemeneti sor */
    char save[MAXLINE]; /* elmentett leghosszabb sor */

    max = 0;
    while ((len = getline(line, MAXLINE)) > 0) {
        printf("%d, %s\n", len, line);
        if (len > max) {
            max = len;
            copy(line, save);
        }
    }
    if (max > 0) /* előfordult sor */
        printf("%s", save);
}

getline(s, lim) /* sor beolvasása s-be, visszaadja a
                hosszát */
char s[];
int lim;
{
    int c, i, j;

    j = 0;
    for (i = 0; (c = getchar()) != EOF && c != '\n'; i++)
        if (i < lim-2) {
            s[j] = c; /* a sor meg a korlatok között */
            ++j;
        }
    if (c == '\n') {
        s[j] = c;
        ++j;
        ++i;
    }
    s[j] = '\0';
    return(i);
}

copy(s1, s2) /* s1 masolása s2-be, ha s2 elég nagy */
char s1[], s2[];
{
    int i;
```



```

    i = 0;
    while ((s2[i] = s1[i]) != '\0')
        ++i;
}

```

Az egyetlen változtatás a fő rutinban:

```
printf("%d, %s\n", len, line);
```

Ez az utasítás kiírja a bemeneti sor hosszát (len) és annyi karaktert, amennyit a line tömbben el lehet menteni.

A getline függvényben néhány módosítást tettünk:

```
for (i=0; (c = getchar()) != EOF && c != '\n'; i++)
```

Az s-ben elmenthető karakterek számának ellenőrzése most már nem a for utasításban történik. Ez a korlát ugyanis már nem feltétele a program befejezésének, mivel getline most már tetszőlegesen hosszú bemeneti sorok hosszát is visszaadja és a szövegből is elment annyit, amennyi lehetséges. Az

```
if (i < lim-2)
```

utasítás dönti el, hogy van-e elég hely a tömbben (még a korlátokon belül). A for ciklusban eredetileg az

```
i < lim-1
```

ellenőrzést használtuk. Ezt azért változtattuk meg, mert az s tömb utolsó indexe

```
lim - 1
```

mivel s-nek lim számú eleme van. Így

```
i < lim-2
```

az esetleges újsor karakternek

```
s[lim-2] = '\n';
```

és a karakterlánc vége jelzőnek

```
s[lim-1] = '\0';
```

helyet hagy. A karakterlánc hossza az i-ben adódik vissza. j tartalmazza az s karakterláncba átmásolt karakterek számát.

#### 1.15. Gyakorlat: (K-R 40. oldal)

Írjunk programot, amely minden olyan sort megjelenít, amely 80 karakternél hosszabb!

```

#include <stdio.h>
#define LONGLINE 80
#define MAXLINE 1000 /* max. bejovo sormeret */

main() /* LONGLINE-nal hosszabb sorok megjelenítése */
{
    int len; /* aktualis sorhossz */
    char line[MAXLINE]; /* aktualis bejovo sor */

    while ((len = getline(line, MAXLINE)) > 0)
        if (len > LONGLINE)
            printf("%s\n", line);
}

```

A program szövegsorokat fogad a bemenetről, amelyek hosszát `getline` adja vissza. Ha a hossz meghaladja a 80 karaktert (`LONGLINE`), akkor a program kiírja a beérkezett sort. Egyébként nem történik intézkedés. Ez az eljárás mindaddig ismétlődik, amíg a program valamennyi bemeneti sort fel nem dolgozta.

A `getline` függvény ugyanaz, mint amit az 1.14. Gyakorlatban láttunk.

#### 1.16. Gyakorlat: (K-R 41. oldal)

Írjunk olyan programot, amely eltávolítja a sorvégi szóközöket és tab-karaktereket a bemenet minden sorából, és törli a teljesen üres sorokat!

```

#include <stdio.h>
#define MAXLINE 1000 /* max. bejovo sormeret */
#define YES 1
#define NO 0

main() /* sorzaro szokoz es tab eltavolito */
{
    char line[MAXLINE]; /* aktualis bejovo sor */

    while (getline(line, MAXLINE) > 0)
        if (remove(line) > 0)
            printf("%s\n", line);
}

remove(s) /* zaro szokozok es tab-ok eltavolitasa s-bol */
char s[];
{
    int i;
    int nl; /* ujsor karakter flag */

    i = 0;
    while (s[i] != '\0') /* megkeresi az s lanc veget */
        ++i;
}

```



```

--i;          /* visszalep 1 pozicioval a '\0'-tol */
if (s[i] == '\n') {
    nl = YES;
    --i;
} else
    nl = NO;
while (i >= 0 && (s[i] == ' ' || s[i] == '\t'))
    --i;
if (i >= 0) {
    ++i;          /* nem ures sor ? */
    /* helyet csinal a '\0'-nak */
    if (nl == YES) {
        s[i] = '\n';
        ++i;
    }
    s[i] = '\0';
}
return(i);
}

```

A `remove` függvény a záró szóközöket és tab-karaktereket távolítja el a line karakterláncból és a lánc új hosszát adja vissza. Ha ez a hossz nagyobb, mint 0, akkor a line a szóköztől és a tabulátortól különböző karaktert is tartalmaz: ekkor a program a sort kiírja. Ellenkező esetben a sor csupán szóközökből és tab-karakterekből áll, és így nem vevődik figyelembe. E mechanizmus gondoskodik arról, hogy a teljesen üres sorok ne íródjanak ki.

A `remove` függvény megkeresi az `s` karakterlánc végét és a `'\0'` (karakterlánc vége jelző) karaktertől eggyel visszapoziционál. Ha a `'\0'` előtti karakter a `'\n'`, az újsor jelző (`nl`) új értéket kap azért, hogy az újsorok a kimeneten is megmaradjanak. A program ezután a szóközőkön és tab-karaktereken keresztül mindaddig visszafelé lépked a karakterláncban, amíg valamilyen más karaktert nem talál, vagy amíg a karakterek el nem fogynak (`i < 0`).

Ha `i >= 0`, akkor a szóközőkön és tab-karaktereken kívül egyéb karakter is előfordult. Ha volt újsor, akkor azt a program visszahelyezi és a karakterlánc végére elhelyezi a `'\0'` jelzőt. Függetlenül attól, hogy `i`-nek mennyi az értéke, a függvény visszaadja `i`-t. A `while` ciklus mindaddig ismétlődik, amíg a program egy üres sort nem talál (`getline 0-t ad vissza`).

A `getline` függvény ugyanaz, mint amit az 1.14. Gyakorlatban láttunk.

#### 1.17. Gyakorlat: (K-R 41. oldal)

Írjunk olyan `reverse(s)` függvényt, amely megfordítja az `s` karakterláncot! Használjuk fel ezt a függvényt olyan program megírásához, amely soronként megfordítja a beolvasott szöveget!

```

#include <stdio.h>
#define MAXLINE 1000 /* max. bemeneti sorhossz */

main() /* a bemenet megfordítása soronként */
{
    char line[MAXLINE]; /* aktualis bemeneti sor */

    while (getline(line, MAXLINE) > 0) {
        reverse(line);
        printf("%s", line);
    }
}

reverse(s) /* az s lanc megfordítása */
char s[];
{
    int i, j;
    char temp;

    i = 0; /* megkeresi az s karakterlanc veget */
    while (s[i] != '\0')
        ++i;
    --i; /* visszalep 1 pozicioval a '\0'-tol */
    if (s[i] == '\n')
        --i;
    j = 0; /* az s karakterlanc kezdete */
    while (j < i) {
        temp = s[j];
        s[j] = s[i]; /* felcsereli a karaktereket */
        s[i] = temp;
        --i;
        ++j;
    }
}

```

A reverse függvény először az s karakterlanc végét keresi meg. A '\0'-tól egyvel visszalep azért, hogy az első karakter ne a karakterlancvég-jelző legyen. Ha '\n' fordul elő, még egy pozícióval visszalep, mivel annak – a '\0'-hoz hasonlóan – a sor végén kell maradnia.

A karakterlanc végének megtalálása után j lesz a lánccs első karakterének indexe, i pedig az utolsó karakteré. Miközben a karakterek helyet cserélnek, j növekszik (a lánccs vége felé halad) és i csökken (a lánccs eleje felé halad). Az eljárás addig tart, amíg j kisebb, mint i.

A main program egyesével olvassa be a bemeneti sorokat, minden sort megfordít és kiírja a megfordított sorokat. A program addig tart, amíg egy üres sor nem fordul elő, azaz

```
getline(line, MAXLINE) == 0
```

A getline függvény ugyanaz, mint amit az 1.14. Gyakorlatban láttunk.



Az 5.4. Gyakorlatban mutatók használatával újraírjuk a reverse függvényt (K-R 111. oldal). A reverse egy másik változatát a rekurzió használatával a 4.9. Gyakorlat ismerteti (K-R 98. oldal).

### 1.18. Gyakorlat: (K-R 43. oldal)

Az előbbi getline függvény for utasításában a feltételvizsgálat meglehetősen ügyetlen. Javítsunk rajta, de úgy, hogy az állomány végén vagy puffertúlcsordulás-kor a program az eddigi módon működjön. Biztos, hogy ez a legjobb szervezés?

```
#include <stdio.h>
#define MAXLINE 1000          /* max. bemeneti sormeret */
#define YES 1
#define NO 0

getline()                     /* módosított változat */
{
    int c, i, okloop;
    extern char line[];

    i = 0;
    okloop = YES;
    while (okloop == YES)
        if (i >= MAXLINE-1)
            okloop = NO;          /* puffer túlcsordulás */
        else {
            c = getchar();
            if (c == EOF || c == '\n')
                okloop = NO;      /* EOF vagy ujsor */
            else {
                line[i] = c;
                ++i;
            }
        }
    }
}
```

A for ciklus helyett egy while ciklust és egy if-else utasítást használunk. Az okloop nevű egész típusú logikai változó vezérli a while ciklus végrehajtását. A ciklus mindaddig ismétlődik, amíg okloop egyenlő nem lesz YES-szel.

A for utasítás második része, vagyis a ciklust vezérlő feltétel most egy if-else utasítás. Ha a három feltétel bármelyike fennáll ( $i \geq \text{MAXLINE}-1$ ,  $c == \text{EOF}$ ,  $c == '\n'$ ), akkor okloop NO értéket kap és a while ciklus véget ér.

Úgy véljük, hogy ez a működés a legésszerűbb. Elérve az állomány végét vagy a puffer túlcsordulásakor a ciklus befejeződik.

### 1.19. Gyakorlat: (K-R 44. oldal)

Írjunk detab néven programot, amely a bemeneten talált tab-karakterek mindegyikét annyi szóközzel helyettesíti, amennyi a következő tabulátorstopig hátra van! Tételezzünk fel egy rögzített tabulátorstop-készletet, a stop-ok mondjuk minden n-edik pozíción találhatók.

```
#include <stdio.h>
#define TABINC 8 /* tab növekmény meret */

main() /* tab-ok helyettesítése szokokokkal */
{
    int c, nb, pos;

    nb = 0; /* a szükséges szokokok száma */
    pos = 1; /* a karakter sorbeli helye */
    while ((c = getchar()) != EOF)
        if (c == '\t') {
            nb = TABINC - (pos - 1) % TABINC; /* tab */
            while (nb > 0) {
                putchar(' ');
                ++pos;
                --nb;
            }
        } else if (c == '\n') {
            putchar(c); /* ujsor */
            pos = 1;
        } else {
            putchar(c); /* minden egyeb */
            ++pos;
        }
}
```

A tabulátorstopok minden TABINC -edik pozíción találhatók. A fenti programban TABINC 8-nak van definiálva. A pos az a szövegsorbeli pozíció, ahol a program éppen tart.

Ha a beolvasott karakter tab, akkor a program kiszámítja az nb értéket, vagyis azt, hogy hány szóközre van szükség a legközelebbi stopig. Az

$$nb = TABINC - (pos-1) \% TABINC;$$

utasítás határozza meg ezt az értéket. Ha a karakter újsor, akkor a program kiírja, és pos ismét a sor elejére inicializálódik (pos = 1). Minden más karakter kiíródik, és pos eggyel növekszik (++pos).

A detab program bővítésére az 5.8. és 5.9. Gyakorlatban kerül sor.



## 1.20. Gyakorlat: (K-R 44. oldal)

Írjuk meg az entab programot, amely a szóközőkből álló karakterláncok helyébe minimális számú tab-karaktert és szóközt ír úgy, hogy a távolság ne változzon! Használjuk ugyanazokat a tabulátor-stopokat, mint a detab-nál.

```
#include <stdio.h>
#define TABINC 8 /* tab növekmény meret */

main() /* szokoz-lanc helyettesítése
        tab-okkal es szokozokkal */
{
    int c, nb, nt, pos;

    nb = 0; /* a szukseges szokozok szama */
    nt = 0; /* a szukseges tab-ok szama */
    for (pos = 1; (c = getchar()) != EOF; pos++)
        if (c == ' ')
            if (pos % TABINC != 0)
                ++nb; /* szokoz behelyezese */
            else {
                nb = 0; /* szokozok helyett tab */
                ++nt;
            }
        else {
            while (nt > 0) {
                putchar('\t'); /* kimenet */
                --nt;
            }
            if (c != '\t')
                while (nb > 0) {
                    putchar(' ');
                    --nb;
                }
            else
                nb = 0;
            putchar(c);
            if (c == '\n')
                pos = 0;
            else if (c == '\t')
                pos = pos + (TABINC - (pos-1) % TABINC) - 1;
        }
}
```

Az nb és nt nevű egész típusú változók tartalmazzák a szóközőkből álló karakterlánc helyettesítéséhez szükséges szóközők és tab-karakterek minimális számát. A pos változó értéke az a szövegsorbeli pozíció, ahol a program éppen tart.

A feladat az összes szóköz megtalálása. A szóközők egy láncát egy tab helyettesíti, ha pos a TABINC többszöröse (pos % TABINC != 0 hamis).

Ha a program egy nem szóköz karakterre talál, akkor kiírja az addig felgyülemlett tab-karaktereket és szókö-

zöket, majd pedig az éppen beolvasott karaktert. Ekkor nb és nt kinullázódik és a program ellenőrzi, hogy nem újsor-karakter érkezett-e. Ha igen, akkor pos ismét az új sor elejére áll.

Ha a beolvasott nemszókőz-karakter egy tab ('\t'), akkor csupán a felgyülemlett tab-karakterek és az éppen beolvasott tab kiírására kerül sor.

Az entab program kibővítésére az 5.8. és 5.9. Gyakorlatban kerül sor.

#### 1.21. Gyakorlat: (K-R 44. oldal)

Írjunk olyan programot, amely a sor n-edik pozíciója előtt előforduló utolsó, nemszókőz-karakter után "összehajtja" a hosszú bemeneti sorokat (n paraméter)! Győződjünk meg róla, hogy a program tényleg ésszerűen működik nagyon hosszú sorok esetén, de akkor is, ha a megadott pozíció előtt egyáltalán nem szerepel szókőz vagy tab!

```
#include <stdio.h>
#define MAXCOL 10          /* max. bemeneti pozicio */

int line[MAXCOL-1];        /* bemeneti sor */

main()                    /* összehajtja a hosszú bemeneti sorokat */
{
    int c, pos;

    pos = 1;              /* sorbeli pozicio */
    while ((c = getchar()) != EOF) {
        line[pos-1] = c;  /* tárolja a kov. karaktert */
        ++pos;
        if (pos >= MAXCOL) {
            pos = findnb(MAXCOL-2); /* nem-szokoz kereses */
            printf(pos);
            pos = newpos(pos, MAXCOL-2);
        } else if (c == '\n') {
            printf(pos-1);
            pos = 1;
        }
    }
}

findnb(col)               /* megkeresi az utolso nem-szokoz
                           poziciojat */
int col;                  /* indulas a col indexen */
{
    while (col >= 0 && (line[col] == ' '
                           || line[col] == '\t'))
        --col;
    return(col+1);
}
```



```

newpos(i, col)                /* megkeresi az új pozíciót */
int col, i;
{
    int j, k;

    if (i > 0) {
        for (j = i, k = 0; j <= col; j++, k++)
            line[k] = line[j];
        return(k+1);
    } else
        return(1);
}

printl(pos)                   /* kinyomtatja a sort */
int pos;
{
    int i;

    for (i = 0; i < pos; i++)
        putchar(line[i]);
    putchar('\n');
}

```

MAXCOL szimbólikus állandó, amely a bemenet n-edik pozícióját jelzi. A pos nevű egész típusú változó kezdeti értéke 1 és arra a szövegsorbeli pozícióra mutat, ahol a program éppen tart. A program a bemeneti sorokat a bemenet n-edik pozíciója előtt törli meg.

Ha

```
if (pos >= MAXCOL)
```

igaz, akkor a program az n-1 -edik pozíción tart, és a bemeneti sort kettétörli.

A findnb függvény a col indexen kezdődő nemszókózt keresi (MAXPOS-2 jelenti a sorbeli utolsó pozíciót). A függvény i+1 -et ad vissza, hogy összhangban legyen a printl függvénnyel. Mivel a bemeneti sor megtörése az n-edik pozíciót megelőző utolsó nemszókóznél történik, a szóköz(öke)t és/vagy tabo(ka)t a következő karakter beolvasása előtt az új sor elejére kell vinni. Ez a newpos feladata. Ha i nagyobb, mint 0, akkor findnb nemszókózkarakterre talált. Bármilyen következzék ezután, a program a legutóbbi nemszókózt a sor elejére viszi:

```

for (j = i, k = 0; j <= col; j++, k++)
    line[k] = line[j];

```

Ha i nem nagyobb, mint 0, akkor line csak szóköz(öke)t és/vagy tabo(ka)t tartalmaz. A program visszaadja az új pos pozíciót.

## 1.22. Gyakorlat: (K-R 44. oldal)

Írjunk olyan programot, amely egy C programból az összes megjegyzést eltünteti! Ne felejtkezzünk meg az idézőjelezett karakterláncok és karakterállandók helyes kezeléséről!

```
#include <stdio.h>

main() /* az osszes megjegyzes eltavolitasa
      ervenyes C programbol*/
{
    int c, d;

    while ((c = getchar()) != EOF)
        if (c == '/')
            if ((d = getchar()) == '*')
                in_comment(); /* megjegyzest talalt */
            else {
                putchar(c); /* nem megjegyzes */
                putchar(d);
            }
        else if (c == '\\' || c == '"')
            echo_quote(c); /* idezojel kezdete */
        else
            putchar(c); /* nem megjegyzes */
}

in_comment() /* ervenyes megjegyzesen belül */
{
    int c, d;

    c = getchar(); /* elozo karakter */
    d = getchar(); /* jelenlegi karakter */
    while (c != '*' || d != '/') { /* megj. veget keresi */
        c = d;
        d = getchar();
    }
}

echo_quote(c) /* idezojel ismetles */
int c;
{
    int d;

    putchar(c);
    while ((d = getchar()) != c) { /* idezojel veg keresese */
        putchar(d);
        if (d == '\\')
            putchar(getchar()); /* escape szekv. atugras */
    }
    putchar(d);
}
```



A program feltételezi, hogy a bemenet egy érvényes C program. A főprogram egy megjegyzés kezdetét (/\*) keresi és ha megtalálta, hívja az in\_comment függvényt. Ez a függvény a megjegyzés végét (\*/) keresi. Az eljárás ily módon biztosítja a megjegyzések elhagyását a C programból.

Ugyancsak a főrutin keresi a szimpla és dupla idézőjeleket és ha talált ilyet, hívja az echo\_quote függvényt. Az echo\_quote argumentuma jelzi, hogy szimpla vagy dupla idézőjelről van-e szó. Az echo\_quote hívásával biztosítjuk, hogy mindazt, amit a program idézőjelek között talál, megismétli a kimeneten és nem téveszti össze a megjegyzéssel. Minden más bemeneti karaktert a program a kimenetre ír. Az echo\_quote nem tekinti záró idézőjelnek a fordított törtvonal (backslash) utáni idézőjelet (lásd az escape szekvenciákról szóló ismertetést a K-R könyv 31. oldalán, valamint az 1.2. Gyakorlatban).

A program akkor ér véget, amikor getchar az állomány végét adja vissza.

### 1.23. Gyakorlat: (K-R 44. oldal)

Írjunk olyan programot, amely egy C programban megtalálja az olyan alapvető szintaktikai hibákat, mint a nem azonos számú nyitó és záró kerek, szögletes, illetve kapcsos zárójelek! Ne feledkezzünk meg az aposztrófokról, idézőjelekről, valamint a megjegyzésekről sem! (Ezt a programot teljes általánosságban nehéz elkészíteni.)

```
#include <stdio.h>

int brace, brack, paren;

main() /* elemi szintaxis vizsgalo */
{
    int c;
    extern int brace, brack, paren;

    while ((c = getchar()) != EOF) {
        if (c == '/') {
            if ((c = getchar()) == '*')
                in_comment(); /* megjegyzesen belül */
            else
                search(c);
        } else if (c == '\\' || c == '"')
            in_quote(c); /* idezojelen belül */
        else {
            search(c);
            if (brace < 0) /* kimeneti hibak */
                printf("Unbalanced braces\n");
            if (brack < 0)
                printf("Unbalanced brackets\n");
            if (paren < 0)
                printf("Unbalanced parentheses\n");
        }
    }
}
```

```

}
if (brace > 0)                /* kimeneti hibak      */
    printf("Unbalanced braces\n");
if (brack > 0)
    printf("Unbalanced brackets\n");
if (paren > 0)
    printf("Unbalanced parentheses\n");
}

search(c)                    /* elemi szintaktikai hibak keresese */
int c;
{
    extern int brace, brack, paren;

    if (c == '{')
        ++brace;
    else if (c == '}')
        --brace;
    else if (c == '[')
        ++brack;
    else if (c == ']')
        --brack;
    else if (c == '(')
        ++paren;
    else if (c == ')')
        --paren;
}

in_comment()                 /* ervenyos megjegyzes belsejeben */
{
    int c, d;

    c = getchar();            /* elozo karakter      */
    d = getchar();            /* jelenlegi karakter  */
    while (c != '*' || d != '/') {
        /* megjegyzes vege keresese */
        c = d;
        d = getchar();
    }
}

in_quote(c)                  /* idezojelen belül      */
int c;
{
    int d;

    while ((d = getchar()) != c)
        /* idezojel veg keresese */
        if (d == '\\')
            /* escape szekv. atugrasa */
            getchar();
}

```

A megoldás nem teljesen általános. A három keresett szintaktikai hiba: a nem azonos számú nyitó és záró kerek, szögletes és kapcsos zárójelek. A program minden egyebet érvényesnek fogad el.



A `search` nevű függvényben a `brace` változó minden alkalommal eggyel növekszik, ha a program nyitó kapcsos zárójelet talál ('{'), és eggyel csökken, ha záró kapcsos zárójelet ('}') fordul elő. Ugyanez történik a `brack` nevű változóval, ha a program nyitó ('[') vagy záró (']') szögletes zárójelet talál, valamint a `paren` nevű változóval, ha a program nyitó ('(') vagy záró (')') kerek zárójelet talál.

A keresés folyamán `brace`, `brack`, illetve `paren` jogosan lehet pozitív vagy nulla értékű. Hibát jelent azonban, ha `brace`, `brack` vagy `paren` bármikor negatívvá válik: ilyenkor a program a megfelelő hibaüzenetet nyomtatja ki. Például `[[[ (brack egyenlő 3)` megengedett egy adott pillanatban, mivel a keresés során a későbbiekben még rátalálhat a program a megfelelő három záró szögletes zárójelre. `]]] (brack egyenlő -3)` azonban nem megengedett, mivel hiányoznak a korábbi nyitó szögletes zárójel-ek, amelyeknek ehhez a három záró szögletes zárójelhez kellene tartozniuk. Ha a megfelelő 3 nyitó szögletes zárójel előfordult volna, akkor `brack`-nek 0-nak kellene lennie. Az

```
if (brace < 0)
    printf("Kapcsos zarojel hiba\n");
if (brack < 0)
    printf("Szogletes zarojel hiba\n");
if (paren < 0)
    printf("Kerek zarojel hiba\n");
```

utasításokra azért van szükség, mert nélkülük a program a `) (` vagy a `]]][[[` vagy a `}}{ {` eseteket nem észlelné hibának.

A `main` rutin a megjegyzéseket valamint a szimpla és dupla idézőjeleket keresi. Ha ilyet talál, akkor mindent, ami belül van, figyelmen kívül hagy. A megjegyzésen és idézetben belüli kapcsos, szögletes és kerek zárójel-ek számának nem feltétlenül kell kiegyenlítettnek lennie.

Miután a program az egész szöveget feldolgozta, még egy utolsó ellenőrzést végez, amelynek során megvizsgálja, nem maradtak-e nyitott kapcsos, szögletes vagy kerek zárójel-ek. Ha igen, akkor az ennek megfelelő üzenetet írja ki.

## 2. FEJEZET

# Típusok, operátorok és kifejezések

---

### 2.1. Gyakorlat: (K-R 50. oldal)

Írjunk az előző for ciklussal egyenértékű ciklust, amely az &&-et nem használja!

Az eredeti ciklus:

```
for(i=0; i<lim-1 && (c=getchar()) != '\n' &&
    c != EOF; ++i)
```

Ennek megfelelője a feladat szerint:

```
for(i=0; (!(i<lim-1) || !((c=getchar()) != '\n') ||
    !(c != EOF)); ++i)
```

Az eredeti ciklusban a feltételes kifejezés a következő:

```
i<lim-1 && (c=getchar()) != '\n' && c != EOF
```

E kifejezés értéke csak akkor IGAZ, ha mindhárom feltétellel értéke IGAZ.

Az && operátor jelentése ES. Ennek megfelelően

```
IGAZ && IGAZ && IGAZ
```

értéke is IGAZ. Ha azonban a feltételek bármelyikének értéke HAMIS, akkor az eredmény is HAMIS.

A || operátor jelentése VAGY. Ennek megfelelően a || operátor eredménye csak akkor HAMIS, ha a feltételek mindegyike HAMIS.

```
HAMIS || HAMIS || HAMIS
```

értéke is HAMIS.

Ennek megfelelően, ha az eredeti for ciklus mindegyik feltételét negáljuk, majd a feltételek között VAGY műveletet írunk elő, azaz

```
(!(i<lim-1) || !((c=getchar()) != '\n') || !(c != EOF))
```

akkor az eredmény csak abban az esetben HAMIS, ha mindegyik feltétel igaz. A végső eredmény negálása IGAZ-at eredményez, jelezve, hogy a fenti két for ciklus egymással egyenértékű.

A megoldást még egy lépéssel továbbfejleszthetjük. Az egyes feltételek negálása megszüntethető, ha az eredeti,



kifejezés ellentétét ellenőrizzük. Egy másik ekvivalens ciklus a következő:

```
for(i=0; !(i>=lim-1 || (c=getchar()) == '\n' ||  
c == EOF); ++i)
```

## 2.2. Gyakorlat: (K-R 54. oldal)

Írjuk meg a `htoi(s)` függvényt, amely egy hexadecimális számjegyekből álló karakterláncot a neki megfelelő egész értékké alakít át! A megengedett számjegyek: 0...9, a...f és A...F.

```
#define YES 1  
#define NO 0  
  
htoi(s) /* az s hexa kar. lanc konvertalasa egeszre */  
char s[];  
{  
    int hexdigit, i, inhex, n;  
  
    n = 0; /* a visszaadando egesz ertek */  
    inhex = YES; /* ervenyes hexa szamjegy */  
    for (i = 0; inhex == YES; i++) {  
        if (s[i] >= '0' && s[i] <= '9')  
            hexdigit = s[i] - '0';  
        else if (s[i] >= 'a' && s[i] <= 'f')  
            hexdigit = s[i] - 'a' + 10;  
        else if (s[i] >= 'A' && s[i] <= 'F')  
            hexdigit = s[i] - 'A' + 10;  
        else  
            inhex = NO; /* nem hexa szamjegy */  
        if (inhex == YES)  
            n = 16 * n + hexdigit;  
    }  
    return(n);  
}
```

A függvény vezérlését a

```
for (i = 0; inhex == YES; i++)
```

utasítás végzi. Az `i` egész az `s` tömb indexe. Mindaddig, amíg `s[i]` érvényes hexadecimális számjegy, az `inhex` értéke YES és a ciklus folytatódik. A `hexdigit` változó értéke 0 és 15 között lehet.

Az

```
if (inhex == YES)
```

utasítás biztosítja, hogy az `s[i]`-ben érvényes hexadecimális számjegy foglaljon helyet, amelynek értéke a `hexdigit`-ben található. Amikor a ciklus befejeződik, a függvény a kiszámított értékkel (`n`) visszatér.

Ez a függvény hasonló az `atoi`-hoz (K-R 51. oldal).

### 2.3. Gyakorlat: (K-R 56. oldal)

Írjuk meg a `squeeze(s1, s2)` egy másik változatát, amely `s1`-ből minden olyan karaktert töröl, amely megegyezik bármelyik `s2` karakterláncbeli karakterrel!

```
squeeze(s1, s2)      /* minden s2-beli karakter torlese
                        s1-bol */
char s1[], s2[];
{
    int i, j, k;

    for (i = k = 0; s1[i] != '\0'; i++) {
        for (j = 0; s2[j] != '\0' && s2[j] != s1[i]; j++)
            ;
        if (s2[j] == '\0') /* lanc vege - nincs egyezes */
            s1[k++] = s1[i];
    }
    s1[k] = '\0';
}
```

Az első utasítás, vagyis

```
for (i = k = 0; s1[i] != '\0'; i++)
```

inicializálja `i`-t és `k`-t, vagyis az `s1` és az eredményül kapott karakterlánc (szintén `s1`) indexét. A program minden olyan `s1`-beli karaktert, amely megegyezik bármelyik `s2`-beli karakterrel, töröl. A program futása az `s1` karakterlánc végéig tart.

A második `for` utasítás minden `s2`-beli karaktert összehasonlít az `s1[i]` karakterrel. Ez a ciklus akkor ér véget, amikor elfogytak az `s2`-beli karakterek vagy pedig ha a program azonos karaktereket talált. Ha nincs egyezés, `s1[i]` átmásolódik a végső karakterláncba. Ha a program egyezést talált, akkor az

```
if (s2[j] == '\0')
```

utasítás nem teljesül és semmi sem történik `s1[i]`-vel – azaz nem másolódik az eredmény-karakterláncba (kihullik).

### 2.4. Gyakorlat: (K-R 56. oldal)

Írjuk meg az `any(s1, s2)` függvényt, amely megadja az `s1` karakterláncnak azt a legelső pozícióját, ahol bármelyik `s2` láncbeli karakter előfordul, és az értéke legyen `-1`, ha `s1` egyetlen `s2`-beli karaktert sem tartalmaz!



```

any(s1, s2)      /* visszaadja az első olyan s1-beli */
char s1[], s2[]; /* pozíciót, ahol bármely s2-beli */
                  /* karakter előfordul */
{
    int i, j;

    for (i = 0; s1[i] != '\0'; i++) {
        for (j = 0; s2[j] != '\0' && s2[j] != s1[i]; j++)
            ;
        if (s2[j] == s1[i])          /* egyezést talált      */
            return(i);              /* az első egyezés helye */
    }
    return(-1);                      /* s1-ben nincs s2-beli karakter */
}

```

A ciklus vezérlését a

```
for (i = 0; s1[i] != '\0'; i++)
```

utasítás végzi. Ha a ciklus normál módon ér véget (elfogynak az s1-beli karakterek), a program -1-gyel tér vissza, jelezve, hogy egyetlen s2-beli karakter sem fordult elő s1-ben.

A második for utasítás, azaz

```
for (j = 0; s2[j] != '\0' && s2[j] != s1[i]; j++)
    ;
```

i minden értékére végrehajtódik. s2 minden karakterét összehasonlítja s1[i]-vel. Ha a ciklus végrehajtása azért ér véget, mert a program egyezést talált (s2 valamelyik karaktere megegyezik az s1 i-edik pozícióján található karakterrel), akkor a program az egyezés pozícióját adja vissza.

## 2.5. Gyakorlat: (K-R 57. oldal)

Módosítsuk a getbits függvényt úgy, hogy a bitpozíciók sorszáma balról jobbra nőjön!

```

getbits(x, p, n)      /* n bitet hoz be a p pozíciótól */
unsigned x, p, n;
{
    return(((x << (p+1-n)) & ~ ((unsigned)~0 >> n))
           >> (wordlength() - n));
}

```

A getbits(x, p, n) függvény jobbra igazítva adja vissza x-nek azt az n-bites mezőjét, amely a p pozíción kezdődik. Az x-beli bitek balról jobbra sorszámozódnak. Tegyük fel például, hogy x 8 bit hosszúságú (az előjel nélküli egészek hossza gépfüggő) és x-nek valamilyen aktuális értéke van:

|   |   |   |   |   |   |   |   |                |
|---|---|---|---|---|---|---|---|----------------|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | bitpozíciók    |
| 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | aktuális érték |

Amennyiben  $n$  és  $p$  értelmes pozitív értékek, például  $n=2$  és  $p=3$ , az

```
x << (p+1-n)
```

kifejezés értéke

```
x << (3+1-2)
x << (2)
```

lesz. Ez a művelet  $x$ -en olyan balra mozgatást végez, hogy az  $n$  bites mező balra fog igazodni:

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|

Az

```
(unsigned) ~0 >> n
```

egy csupa egyesből álló egész szám bitjeit  $n$  pozícióval jobbra lépteti. Ez által a bal szélén egy  $n$  bites, csupa 0-ból álló mező keletkezik. Ezen érték 1-es komplemente ( $\sim$ ) egy  $n$  bites, csupa egyesekből álló mezőt hoz létre a bal oldalon, míg csupa 0 bitet a többi pozíción. Amennyiben ezt az értéket logikai ES kapcsolatba hozzuk az elmozgatott  $x$ -szel, balra igazítva megkapjuk  $x$ -nek a  $p$  pozíción kezdődő  $n$  bites mezőjét.

|       |   |   |   |   |   |   |   |                              |
|-------|---|---|---|---|---|---|---|------------------------------|
| 0     | 1 | 1 | 0 | 1 | 1 | 0 | 0 | a léptetett x                |
| 1     | 1 | 0 | 0 | 0 | 0 | 0 | 0 | a kifejezés értéke           |
| ----- |   |   |   |   |   |   |   |                              |
| 0     | 1 | 0 | 0 | 0 | 0 | 0 | 0 | balra igazított n bites mező |

Ha a feladat az, hogy az  $n$  bites mező jobbra igazítva jelenjen meg, akkor egy új értékkel kell az  $n$  bites mezőt jobbra mozgatni: ez az érték az előjeltelen egészben található bitek száma mínusz  $n$ . A `wordlength` függvény megadja az előjeltelen egészben található bitek számát. `wordlength` leírása a 2.6. Gyakorlatban található.

## 2.6. Gyakorlat: (K-R 57. oldal)

Írjunk olyan `wordlength()` függvényt, amely kiszámítja a befogadó gép szóhosszúságát, azaz meghatározza, hogy egy int mennyiségben hány bit van! A függvény legyen gépfüggetlen, vagyis a forráskód minden gépen működjön!

```
wordlength() /* megadja a befogadó gép szóhosszat */
{
    int i;
    unsigned v = ~0;
```



```

    for (i = 1; (v = v >> 1) > 0; i++)
        ;
    return(i);
}

```

A *v* előjel nélküli egész változót csupa 1-esre inicializáljuk:

```
unsigned v = ~0
```

Az *i* változóban azt számláljuk, hányszor kell jobbra léptetni, mire *v* csupa nullából fog állni.

*v*-t eggyel jobbra toljuk és a kapott értéket rendeljük *v*-hez:

```
(v = v >> 1) > 0
```

Ezután ellenőrizzük, hogy az új érték nagyobb-e, mint 0. Ha *v* bármelyik bitpozícióján 1 áll, az eredmény igaz. A ciklus mindaddig folytatódik, amíg *v* minden bitjét jobbra toltuk; *i* tartalmazza a szükséges léptetések darabszámát, ami a szóhosszúsággal azonos.

## 2.7. Gyakorlat: (K-R 57. oldal)

Írjunk olyan `rightrot(n, b)` függvényt, amely *b* számú bitpozícióval jobbra történő bitrotációt végez az *n* egész típusú mennyiségen!

```

rightrot(n, b)    /* n jobbra forgatasa b pozicioval */
int n, b;
{
    int rbit;
    int lshift = wordlength() - 1;
    int clmbit = (unsigned) ~0 >> 1;

    while (b-- > 0) {
        rbit = n & 1;                /* jobb szelso bit      */
        n = (n >> 1) & clmbit;        /* n jobbra lep 1-gyel */
        rbit = rbit << lshift;        /* bal szelso bit torlese */
        n = n | rbit;                /* a rotacio befejezese */
    }
    return(n);
}

```

Az *rbit* változó tartalmazza *n* jobb szélső bitjét. *lshift* a jobb szélső bit balra léptetésének számlálója, míg *clmbit* (bal szélső bit törölő) tartalma olyan bitmező, amely csupa egyesnek eggyel jobbra történő mozgatásával áll elő.

n jobbra léptetése és a bal szélső bit törlése az

```
n = (n >> 1) & clmbit;
```

utasítás feladata - ezzel a program gondoskodik arról, hogy az előjelbit nehegy feltámadjon olyan gépek esetében, amelyek aritmetikai jobbra léptetést végeznek. A következő lépésben

```
rbit = rbit << lshift;
```

a jobb szélső bitet a bal szélső pozícióra mozgatja. Ha ezt az értéket VAGY kapcsolatba hozzuk az n jobbra mozgatott értékével, megtörténik egy rotáció. b db rotáció hajtódik végre az n értéken.

```
rightrot(n, b)
int n, b;
{
    int rbits;

    if ((b = b % wordlength()) > 0) {
        rbits = ~(~0 << b) & n; /* n jobb old. b bitje */
        n = n >> b;             /* n jobbra tolása b-vel */
        if (n < 0)              /* n negatív szám ? */
            n = n & ((unsigned) ~0 >> b);
                                /* újra belepo bit torlese */
        rbits = rbits << (wordlength() - b);
                                /* jobb oldali b bit balra */
        n = n | rbits;          /* forgatas befejezve */
    }
    return(n);
}
```

Íme egy másik megoldás ugyanarra a feladatra. Ha a jobbra rotálandó pozíciók száma (b) ugyanannyi, mint az egész típusú mennyiségben található bitek száma, semmi sem változik, mivel n ugyanannyi, mint a rotálás előtt. Ha b ennél kevesebb, csak b számú pozíciót kell rotálni. Ha b meghaladja az egészbeli bitek számát, csak a b-nek a szóhosszal történő elosztásakor keletkező maradék (moduló operátor) lesz a szükséges rotálások száma. E megfontolásból következően nincs szükség ciklusra.

```
~0 << b      az összes egyes b pozícióval balra
              tolódik és b számú nulla marad a
              bal szélső pozíciókon
~(~0 << b)   az összes egyes a b számú jobb
              szélső pozíción van
```

Ha ezt az értéket bitenkénti ES kapcsolatba hozzuk n-nel, akkor az n mennyiség b számú jobb szélső bitje rendelődik rbit-hez. Ezután lehet n-et b pozícióval jobbra mozgatni. Ha n negatív szám, az előjelbit újra megjelenhet olyan gépek esetében, amelyek aritmetikai jobbra mozgatást végeznek, ezért törölni kell. Az rbits bitek ezután a bal szélső pozícióra mozdulnak. A program



ezt az értéket VAGY kapcsolatba hozza az  $n$  mennyiség jobbra mozgatott értékével, amivel befejeződik az  $n$  egész szám  $b$  pozícióval történő jobbra rotálása.

## 2.8. Gyakorlat: (K-R 57. oldal)

Írjunk olyan  $\text{invert}(x, p, n)$  függvényt, amely az  $x$ -ben a  $p$  pozíciótól kezdve  $n$  bitet invertál (vagyis az 1-eseket 0-ra, a 0-kat 1-esekre cseréli fel), miközben a többi bit változatlan marad!

```
invert(x, p, n)      /* x n bitjének invertálása p
                        pozíciótól */
unsigned x, p, n;
{
    return(x ^ (~0 << n) << (p+1-n));
}
```

Elsőnek

$(\sim 0 \ll n)$

az összes egyest  $n$  pozícióval balra mozgatja  $n$  darab nullát hagyva a jobb szélső pozíciókon.

Másodszor,

$\sim(\sim 0 \ll n)$

csupa egyeseket helyez el a jobb szélső pozíciókon és nullákat az összes többi pozíción.

Harmadszor,

$(\sim(\sim 0 \ll n) \ll (p+1-n))$

ezt az  $n$  darab 1-es bitet a  $p$  pozícióba mozgatja.

Végezetül,

$x \wedge (\sim(\sim 0 \ll n) \ll (p+1-n))$

a bitenkénti kizáró VAGY operátor ( $\wedge$ ) 1-et eredményez, ha a két bit különböző és 0-t, ha azonosak. Mivel a feladat a  $p$  pozíción kezdődő  $n$  bit invertálása, elegendő, ha kizáró VAGY kapcsolatba hozzuk ezeket a biteket a  $p$  pozíciótól kezdődő csupa 1-es bitekkel  $n$  biten keresztül (a többi pozíción 0 bitek vannak). Ha egy eredeti bit értéke 0, amit kizáró VAGY kapcsolatba hozunk az 1 bittel, akkor az eredmény 1, vagyis az eredeti bit átfordult. Ha egy eredeti bit értéke 1, amit kizáró VAGY kapcsolatba hozunk az 1 bittel, akkor az eredmény 0, vagyis az eredeti bit ugyancsak átfordult.

Az  $n$  bites mezőn kívüli pozíciók a nullákkal kerülnek kizáró VAGY kapcsolatba:  $0 \wedge 0$  (azonos bitek) 0-t ered-

ményez - semmi nem változik, míg  $1 \wedge 0$  (különböző bitek) 1-et ad, vagyis megint csak nincs változás.

## 2.9. Gyakorlat: (K-R 59. oldal)

2-es komplementű aritmetikában  $x \& (x-1)$  törli  $x$  jobb oldali szélső 1-es bitjét. (Miért?) Kihasználva ezt a megfigyelést, írjuk meg a bitcount egy gyorsabb változatát!

```
bitcount(n)    /* az 1 bitek szama n-ben -  
                                     gyorsabb változat */  
unsigned n;  
{  
    int b;  
  
    for (b = 0; n != 0; n &= n-1)  
        ++b;  
    return(b);  
}
```

Adjunk valamilyen értéket  $x-1$ -nek, legyen például az 1010 bináris szám, amelynek decimális értéke 10.  $(x-1)+1$  azonos  $x$ -szel:

| bináris |       | decimális |
|---------|-------|-----------|
| 1010    | $x-1$ | 10        |
| + 1     |       | + 1       |
| -----   |       | -----     |
| 1011    |       | 11        |

A bináris összeadást tekintve az összeadás egyik következménye, hogy az  $x-1$  jobb szélső 0 bitje az  $x$  eredményben 1-esre változik. Ennek megfelelően az  $x$  jobb szélső 1-es bitjének az  $x-1$ -ben egy 0 bit felel meg. Ez a magyarázata annak, hogy a 2-es komplementű számrendszerben  $x \& (x-1)$  törli az  $x$  jobb szélső 1-es bitjét.

Tekintsünk egy négybites előjel nélküli mennyiséget. Ahhoz, hogy e mennyiség 1-es bitjeit megszámlálja, az eredeti bitcount négy mozgatást végez a jobb szélső bit ellenőrzése végett. Alternatív megoldásként kihasználhatjuk azt a tényt, hogy  $x \& (x-1)$  törli az  $x$  jobb szélső 1-es bitjét. Ha például  $x$  értéke 9, akkor

|         |                           |
|---------|---------------------------|
| 1 0 0 1 | bináris 9 érték ( $x$ )   |
| 1 0 0 0 | bináris 8 érték ( $x-1$ ) |
| -----   |                           |
| 1 0 0 0 | $x \& (x-1)$              |

és ezáltal  $x$  jobb szélső 1-es bitje törlődött. Az eredmény a bináris 1000, azaz a 8 decimális mennyiség. Az eljárást megismételve



|       |   |   |   |                       |
|-------|---|---|---|-----------------------|
| 1     | 0 | 0 | 0 | bináris 8 érték (x)   |
| 0     | 1 | 1 | 1 | bináris 7 érték (x-1) |
| ----- |   |   |   |                       |
| 0     | 0 | 0 | 0 | x & (x-1)             |

aminek következtében x jobb szélső 1-es bitje törölődött. Az eredmény a bináris 0000, azaz a 0 decimális mennyiség. Miután x nem tartalmaz több 1-est, az eljárás befejeződik.

A legrosszabb eset az, amikor x csupa 1-esből áll - ebben az esetben az ES-ek száma ugyanannyi, mint a mozgatók száma a bitcount-ban. Általában azonban ez a gyorsabb változat.

## 2.10. Gyakorlat: (K-R 60. oldal)

Irjuk át a lower függvényt, amely a nagybetűs karaktereket kisbetűsökké konvertálja! Az if-else helyett használjunk feltételes kifejezést!

```
lower(c)    /* c konvertalasa kisbetusse: csak ASCII */
int c;
{
    return(c >= 'A' && c <= 'Z' ? c + 'a' - 'A' : c);
}
```

Amennyiben a

```
c >= 'A' && c <= 'Z'
```

kifejezés igaz, akkor c nagybetűs karakter (csak ASCII). Ebben az esetben a

```
c + 'a' - 'A'
```

kifejezés kiértékelődik és a program a kisbetűs karaktert adja vissza. Egyébként a karakter változtatás nélkül adódik vissza.

### 3. FEJEZET

## Vezérlési szerkezetek

---

#### 3.1. Gyakorlat: (K-R 67. oldal)

Írjuk meg az `expand(s, t)` függvényt, amely – miközben az `s` karakterláncot a `t` karakterláncba másolja – a láthatatlan karaktereket (pl. újsor és tab) látható escape szekvenciákká (pl. `\n` és `\t`) alakítja át! Használjuk a `switch` utasítást!

```
expand(s, t) /* újsor és tab átalakítása láthatóvá */
char s[], t[];
{
    int i, j;

    for (i = j = 0; s[i] != '\0'; i++)
        switch (s[i]) {
            case '\n': /* újsor */
                t[j++] = '\\';
                t[j++] = 'n';
                break;
            case '\t': /* tab */
                t[j++] = '\\';
                t[j++] = 't';
                break;
            default: /* minden más karakter */
                t[j++] = s[i];
                break;
        }
    t[j] = '\0';
}
```

A

```
for (i = j = 0; s[i] != '\0'; i++)
```

utasítás végzi a ciklus vezérlését. Az `i` változó az eredeti `s` karakterláncban futó index, míg `j` az átmásolt és módosított `t` karakterlánc indexe.

A `switch` utasításban három eset (`case`) van: `'\n'`, vagyis az újsor-karakter; `'\t'`, vagyis a tab-karakter valamint a `default` (egyéb) eset. Ha az `s[i]` karakter nem azonos az első két eset valamelyikével, akkor a `default` címkéjű `case` hajtódik végre: `s[i]` egyszerűen átmásolódik a `t` karakterláncba.

#### 3.2. Gyakorlat: (K-R 70. oldal)

Írjuk meg az `expand(s1, s2)` függvényt, amely az `s1` karakterláncban található rövidítéseket `s2`-ben teljes



listává bővíti ki (pl. a-z helyett abc...xyz-t ír)! Engedjük meg a kis- és nagybetűket, illetve a számjegyeket is, és készüljünk fel az olyan esetek kezelésére is, mint a-b-c és a-z0-9 és -a-z! (Hasznos megállapodás, ha a vezető- és záró-karaktereket betű szerint vesszük.)

```
expand(s1, s2) /* rovidített jelölés kiterjesztése */
char s1[], s2[];
{
    int err, i, j, last;
    int from, to;

    err = 0; /* nem talált hibát */
    j = 0; /* kiterjesztett lanc indexe */
    last = strlen(s1) - 1; /* utolso pozicio indexe */
    for (i = 0; i <= last; i++) {
        if (s1[i] == '-') /* - karakter */
            if (i == 0 || i == last) /* nyitó vagy záró - */
                from = to = '-';
            else if (to != '-') { /* a-b-c eset */
                from = to + 1;
                if (s1[i+1] != '-') /* ne legyen két - */
                    to = s1[i+1]; /* egymás után */
                else {
                    to = -1; /* megjegyzi a hibát */
                    ++i; /* nem kezeli a rovidítést */
                }
            } else { /* az utolsó karakter - volt */
                to = -1; /* feljegyzi a hibát */
                ++i;
            }
        else { /* nem - karakter */
            from = s1[i];
            if (i+2 > last) /* nincs elég kar. */
                to = -1;
            else if (s1[i+1] != '-') /* - karaktert várt */
                to = -1;
            else if (s1[i+2] == '-') /* ne legyen két - */
                to = -1; /* egymás után */
            else { /* érvényes rovidítésnek látszik */
                i += 2;
                to = s1[i];
            }
        }
        if (from > to) /* előfordult hiba ? */
            err = -1;
        else /* kiterjeszti a rovidítést */
            for (; from <= to; from++)
                s2[j++] = from;
    }
    s2[j] = '\0';
    return(err);
}
```

Az `strlen` függvény egy karakterlánc hosszát adja vissza (K-R. 48. oldal). A karakterlánc utolsó karakterének indexe egyenlő a karakterlánc hossza mínusz 1-gyel:

```
last = strlen(s1) - 1;
```

Az aktuális `s1[i]` karakter a '-' is lehet. Amennyiben a '-' karakter a lánc első vagy utolsó karaktere, úgy a program azt betű szerint értelmezi:

```
if (i == 0 || i == last)
    from = to = '-';
```

Ha viszont nem az első vagy az utolsó karakter, akkor a program az előző karaktert is ellenőrzi annak érdekében, hogy ne fordulhasson elő két '-' karakter egymás után:

```
else if (to != '-')

```

Ez történik például az olyan karakterláncban, mint `a-b-c`, miután a program `a-b-t` azonosította. Az

```
if (s1[i+1] != '-')

```

utasítás megint ellenőrzi a kettős - esetet annak érdekében, hogy elkerüljük a nem megengedett rövidítéseket, mint pl. `a-b--c`.

A másik eset, amikor az aktuális karakter nem '-'. Ilyenkor még legalább két további karakterre van szükség: egy '-' karakterre, amit valamilyen más karakter követ. Ha tehát a következő karakter nem '-', akkor hiba van:

```
else if (s1[i+1] != '-')
    to = -1;
```

Ha a harmadik karakter nem '-', akkor a program olyan szekvenciát talált, mint pl. `a-z`.

A `from` és `to` változók tartalmazzák a rövidített jelölésmód karaktereit. Hibát jelent, ha a `from` változó értéke nagyobb, mint a `to` változóé (mint pl. a `z-a` esetben).

Ha a jelölésmód érvényes, akkor megtörténik az `s2`-be való kibővítés:

```
for (; from <= to; from++)
    s2[j++] = from;
```

Ha a program a jelölésmódban bárhol hibát észlel, a `to` változónak -1 értéket ad. Ez az érték mindig kisebb, mint a `from` változó értéke és ennek hatására a program az aktuális jelölésmódot figyelmen kívül hagyja. Az `err` változónak -1 értéket ad, hogy ezzel emlékeztessen a hibára; ha pedig nem fordul elő hiba, akkor az `err` értéke 0. A program visszaadja az `err` változót, ezzel jelezve a sikertelen vagy sikeres végrehajtást.

A feladat leírásában betűk és számok szerepelnek. Két lehetőség van: vagy úgy írjuk meg a programot, hogy az a betűket, számokat és a '-' karaktereket terjessze ki,



vagypedig mindenféle karakterre elvégzi a bővítést. Minthogy egyszerűbb a tetszőleges karakterkészletre történő bővítés (amely a betűket, számokat és a '-' karaktert is tartalmazza), a fenti megoldás a második lehetőség szerint jár el. Az expand függvény ASCII karakterekkel működik. Az a-z rövidítés a neki megfelelő abc...xyz listává bővül ki. A !-~ rövidítés kiterjesztésének eredménye !"..ABC..XYZ..abc..xyz ..!}~ ami nem akármilyen.

### 3.3. Gyakorlat: (K-R 71. oldal)

Kettes komplementű számábrázolásban az itoa függvény általunk írt változata nem kezeli a legnagyobb negatív számot, tehát a  $-(2^{\text{száméret}-1})$  értékű n-et. Magyarázzuk meg, hogy miért! Módosítsuk úgy a programot, hogy ezt az értéket helyesen írja ki, függetlenül attól, hogy milyen gépen fut!

```
#define abs(x) ((x) < 0 ? -(x) : (x))

itoa(n, s) /* n karakterekre konvertálása s-be
            - módosított változat */
char s[];
int n;
{
    int i, sign;

    sign = n; /* feljegyzzi az előjelet */
    i = 0;
    do { /* számjegyek generalása ford. sorrendben */
        s[i++] = abs(n % 10) + '0'; /* kov. számjegy */
    } while ((n /= 10) != 0); /* torles */
    if (sign < 0)
        s[i++] = '-';
    s[i] = '\0';
    reverse(s);
}
```

száméret-1  
A  $-(2^{\text{száméret}-1})$

nem konvertálható pozitív számmá úgy, mint

n = -n;

mivel a 2-es komplementű számábrázolásban a legnagyobb pozitív szám

száméret-1  
2 - 1

A sign nevű változó tartalmazza n kezdeti értékét. Az abs nevű makró állítja elő n % 10 abszolút értékét. Ezáltal a program elkerüli a

```
    szómeret-1
  -(2      )
```

tipusú problémát, mivel a program csupán a moduló művelet eredményét teszi pozitívvá. A do-while utasításban található feltételes kifejezés az eredeti

```
(n /= 10) > 0
```

alakról az

```
(n /= 10) != 0
```

alakra változott, mivel, ha  $n$  negatív, akkor negatív marad az itoa végrehajtása során.

### 3.4. Gyakorlat: (K-R 71. oldal)

Írjuk meg a hasonló itob( $n$ ,  $s$ ) függvényt, amely az  $n$  előjel nélküli (unsigned) egész számot bináris karakterábrázolásban az  $s$  karakterláncba konvertálja! Írjuk meg az itoh függvényt is, amely egy egész számot hexadecimális ábrázolásmódba alakít át!

```
itob(n, s)      /* n karakterekke konvertalasa s-be -
                  binaris változat */
char s[];
unsigned n;
{
    int i;

    i = 0;
    do {      /* szamjegyek generalasa ford. sorrendben */
        s[i++] = n % 2 + '0';      /* kovetkezo szamjegy */
    } while ((n /= 2) != 0);      /* torli */
    s[i] = '\0';
    reverse(s);
}
```

Minthogy  $n$  előjel nélküli, nincs szükség az előjel feljegyzésére.  $n$  tartalma binárisra konvertálódik, így

```
n % 2
```

vagy 0-t, vagypedig 1-et eredményez, és

```
n /= 2
```

törli ezt a számot  $n$ -ből. Az iteráció addig folytatódik, amíg  $n/2$  0 nem lesz.



```

itoh(n, s)      /* n karakterekke konvertalasa s-be -
                  hexadecimalis változat */
char s[];
unsigned n;
{
    int h, i;

    i = 0;
    do {          /* számjegyek generalasa ford. sorrendben */
        h = n % 16;          /* kovetkezo számjegy */
        s[i++] = (h <= 9) ? h+'0' : h+'a'-10;
    } while ((n /= 16) != 0); /* torli          */
    s[i] = '\0';
    reverse(s);
}

```

Az itoh függvény hasonló a gyakorlat első részében lá-  
tott itob-hoz. Az egyik különbség az, hogy

```
h = n % 16;
```

0 és 15 közötti számokat ad vissza a 0-t és a 15-öt is  
beleértve.

Az

```
s[i++] = (h <= 9) ? h+'0' : h+'a'-10;
```

utasítás hexadecimalis karaktert ad vissza. Ha h kisebb  
vagy egyenlő 9-cel, akkor az ASCII karakter

```
h + '0'
```

lesz. Egyébként pedig

```
h + 'a' - 10
```

egy a és az f közötti kisbetűt állít elő, amely a 10 és  
15 közötti értékeket ábrázolja (inkluzív).

### 3.5. Gyakorlat: (K-R 71. oldal)

Írjuk meg az itoa függvénynek azt a változatát, amely  
kettő helyett három argumentumot fogad! A harmadik argu-  
mentum a minimális mezőszélesség; az átkonvertált számot  
szükség esetén balról üres közökkel kell kitölteni, hogy  
elég széles legyen.

```

#define abs(x) ((x) < 0 ? -(x) : (x))

itoa(n, s, w) /* n karakterekke konvertalasa s-be, */
char s[];     /* w karakter szélességben          */
int n, w;
{
    int i, sign;

    sign = n; /* feljegyzi az elojelet */
}

```

```

i = 0;
do {      /* szamjegyek generalasa ford. sorrendben */
    s[i++] = abs(n % 10) + '0';      /* kov. szamjegy */
} while ((n /= 10) != 0);            /* torli */
if (sign < 0)
    s[i++] = '-';
while (i < w) /* szokozokkal tolt, amig van hely */
    s[i++] = ' ';
s[i] = '\0';
reverse(s);
}

```

A függvény hasonló a 3.3. Gyakorlatban látott itoa függvényhez. A szükséges módosítás:

```

while (i < w)
    s[i++] = ' ';

```

A while ciklus szükség esetén szóközökkel egészíti ki a karakterláncot.

### 3.6. Gyakorlat: (K-R 73. oldal)

Írjunk olyan programot, amely a bemenetét a kimenetére másolja, de ha a bemenetre egymás után többször érkezik ugyanaz a sor, akkor azt csak egyszer nyomtatja ki! (Ez egyszerű változata a UNIX uniq szolgáltatásának.)

```

#include <stdio.h>
#define MAXLINE 100

main() /* egymás utáni azonos sorok torlese - uniq */
{
    char line[MAXLINE];      /* aktualis bemeneti sor */
    char lastline[MAXLINE]; /* legutobbi bemeneti sor */

    lastline[0] = '\0';
    while (getline(line, MAXLINE) > 0)
        if (strcmp(line, lastline) != 0) {
            printf("%s\n", line);
            copy(line, lastline);
        }
}

```

A getline függvény ugyanaz, mint amit az 1.14. Gyakorlatban használtunk.

A lastline karakterlánc tartalmazza az utoljára beolvasott sort. A program indulásakor a lastline üres. strcmp két karakterláncot hasonlít össze és 0-t ad vissza, ha a két lánc azonos. strcmp leírása a K-R könyv 110. oldalán található, míg a copy függvényé a K-R könyv 39. oldalán.



```

#include <stdio.h>
#define MAXLINE 100

main() /* egymás utáni azonos sorok torlese - uniq */
{
    char line[MAXLINE]; /* aktualis bemeneti sor */
    char lastline[MAXLINE]; /* legutobbi bemeneti sor */

    lastline[0] = '\0';
    while (getline(line, MAXLINE) > 0) {
        if (strcmp(line, lastline) == 0)
            continue;
        printf("%s\n", line);
        copy(line, lastline);
    }
}

```

A fenti megoldás a `continue` utasítást használja. Amennyiben a `line` és `lastline` karakterláncok azonosak, a program a végrehajtást a `while` utasítás ellenőrző részével folytatja:

```

if (strcmp(line, lastline) == 0)
    continue;

```

Egyéb esetben a `printf` és a `copy` hajtódik végre és csak ezt követően kerül sor ismét a `while` utasítás végrehajtására.

## 4. FEJEZET

# Függvények és programstruktúra

---

### 4.1. Gyakorlat: (K-R 78. oldal)

Írjunk egy `rindex(s, t)` függvényt, amely `t` `s`-beli jobb-  
oldali legszélső előfordulásának pozícióját adja vissza,  
illetve `-1`-et ad, ha `t` nem fordul elő `s`-ben!

```
rindex(s, t)          /* t jobb szelso indexe s-ben,  
                        -1 ha nincs */  
char s[], t[];  
{  
    int i, j, k, pos;  
  
    pos = -1;          /* t indexe s-ben */  
  
    for (i = 0; s[i] != '\0'; i++) {  
        for (j=i, k=0; t[k]!='\0' && s[j]==t[k]; j++, k++)  
            ;  
        if (t[k] == '\0')  
            pos = i;  
    }  
    return(pos);  
}
```

Az `rindex` hasonló az `index` rutinhoz (K-R 77. oldal). Ha a program az `index`-ben egyezést talál, a `t` első elemének `s`-beli pozícióját adja vissza a hívó programnak. Az `rindex` esetében viszont ez a pozíció a `pos` változóban tárolódik:

```
if (t[k] == '\0')  
    pos = i;
```

és a keresés `s[i+1]`-en folytatódik. Végül, ha a program az `s` tömböt végigkereste (azaz `s[i]` értéke `'\0'`), a program `t` `s`-beli jobb oldali legszélső előfordulásának pozícióját adja vissza.

### 4.2. Gyakorlat: (K-R 81. oldal)

Bővítsük ki `atof`-ot oly módon, hogy az

123.45e-6

alakú tudományos jelölésmódot is kezelni tudja, ahol a lebegőpontos számot `e` vagy `E` és egy esetleges előjellel ellátott kitevő is követheti!



```

double atof(s) /* az s lanc konvertalasa double-la */
char s[];      /* kezeli a fellogaritmikus jelolest */
{
    double val, power;
    int exp, i, sign;

    for(i=0; s[i]!=' ' || s[i]!='\n' || s[i]!='\t'; i++)
        ; /* atlepi a kozeket */
    sign = 1;
    if (s[i] == '+' || s[i] == '-') /* elojel */
        sign = (s[i++] == '+') ? 1 : -1;
    for (val = 0; s[i] >= '0' && s[i] <= '9'; i++)
        val = 10 * val + s[i] - '0'; /* egesz resz */
    if (s[i] == '.')
        i++;
    for (power = 1; s[i] >= '0' && s[i] <= '9'; i++) {
        val = 10 * val + s[i] - '0'; /* tortresz */
        power *= 10;
    }
    val = sign * val / power;

    if (s[i] == 'e' || s[i] == 'E') {
        i++; /* tudomanyos jelolesmod */
        sign = 1;
        if (s[i] == '+' || s[i] == '-')
            sign = (s[i++] == '+') ? 1 : -1;
        for (exp = 0; s[i] >= '0' && s[i] <= '9'; i++)
            exp = 10 * exp + s[i] - '0';
        if (sign == 1)
            while (exp-- > 0) /* pozitiv kitevo */
                val *= 10;
        else
            while (exp-- > 0) /* negativ kitevo */
                val /= 10;
    }
    return(val);
}

```

Az `exp` egesz tipusú változó tartalmazza a kitevő abszolút értékét. A `sign` változó előbb a mantissza előjelét tartalmazza, majdpedig a kitevőét - feltéve, hogy van kitevő. A `power` `double` tipusú változó egyenlő 10-nek a tizedesponttól jobbra eső számjegyek darabszámára emelt hatványával. A `val` `double` tipusú változóban először a mantissza értéke, majd a teljes számé (mantissza és kitevő) található.

A rutin első fele az `atof` megismétlése (K-R 80. oldal). A program a láthatatlan közöket átugorja, az előjelet feljegyzí és a mantisszát eltárolja a `val` változóban. Ezután azonban, ahelyett, hogy azonnal visszatérne az eredménnyel (`sign * val / power`), ezt az értéket a `val` változóban tárolja.

A program második fele az esetleges kitevőt dolgozza fel. Ha nincs kitevő, akkor a `val`-ban tárolt szám visszaadódik a hívó programnak. Ha viszont van kitevő, akkor a program a kitevő előjelét tárolja a `sign` válto-

zóban, kiszámítja a kitevő értékét és az exp változóban tárolja.

A záró művelet, vagyis

```
if (sign == 1)
    while (exp-- > 0)
        val *= 10;
else
    while (exp-- > 0);
    val /= 10;
```

módosítja a mantisszát a kitevő értékének függvényében. Ha a kitevő pozitív, a program exp alkalommal 10-zel megszorozza a mantisszát. Ha a kitevő negatív, a program exp alkalommal 10-zel elosztja a mantisszát. Ezek után val a végső, teljes számot tartalmazza, ami a hívó programnak adódik vissza.

A program 10-zel elosztja val-t ahelyett, hogy 0.1-gyel megszorozná, mivel 0.1 binárisan nem pontos tört. A legtöbb gépen 0.1 úgy van ábrázolva, hogy az érték valamivel kevesebb 0.1-nél, ezért  $10.0 \cdot 0.1$  ritkán egyenlő 1.0-val. Az ismételt osztás 10-zel előnyösebb, mint az ismételt szorzás 0.1-gyel, noha még azzal is veszítünk bizonyos mértékig a pontosságból.

#### 4.3. Gyakorlat: (K-R 86. oldal)

Az alapvető programkeret megtartásával egyszerűen kibővíthetjük a kalkulátorprogramot. Vezessük be a moduló (%) és az egyoperandusú mínusz operátorokat! Vezessük be továbbá az erase parancsot, amely törli a verem legfelső elemét! Vezessünk be változónevek kezelését lehetővé tevő parancsokat! (A huszonhat egybetűs változó névre ez egyszerűen megoldható.)

```
#include <stdio.h>

#define MAXOP 20 /* operandus, operator max. meret */
#define NUMBER '0' /* jelzi, hogy számot talált */
#define TOOBIG '9' /* jelzi, hogy a lanc túl hosszú */

double storage[26]; /* a 26 változó neve (A-Z) értéke */

main() /* fordított lengyel elvű asztali kalkulátor */
{
    int type, var = 0;
    char s[MAXOP];
    double op2, atof(), pop(), push();

    while ((type = getop(s, MAXOP)) != EOF) {
        switch (type) {
            case NUMBER:
                push(atof(s));
                break;
```



```

case '+':
    push(pop() + pop());
    break;
case '*':
    push(pop() * pop());
    break;
case '-':
    op2 = pop();
    push(pop() - op2);
    break;
case '/':
    op2 = pop();
    if (op2 != 0.0)
        push(pop() / op2);
    else
        printf("nulla osztó leemelve\n");
    break;
case '%':
    op2 = pop();
    if (op2 != 0.0)
        push((double) ((int)pop() % (int)op2));
    else
        printf("nulla osztó leemelve\n");
    break;
case '=':
    printf("\t%f\n", push(pop()));
    break;
case 'c':
    clear();
    break;
case 'e':
    erase();
    break;
case 'm':
    push(-pop());
    break;
case 's':
    if (var >= 'A' && var <= 'Z') {
        pop();
        storage[var - 'A'] = pop();
    } else
        printf("nincs tarolando változó\n");
    break;
case TOOBIG:
    printf("%.20s ... túl hosszú\n", s);
    break;
default:
    if (type >= 'A' && type <= 'Z')
        push(storage[type - 'A']);
    else
        printf("ismeretlen parancs: %c\n", type);
    break;
}
var = type;

```

```

erase()    /* torli a veremrol a legfelső bejegyzést */
{
    extern int sp;          /* veremmutató */
    if (sp > 0)
        --sp;
    else {
        printf("hiba: a verem üres\n");
        clear();
    }
}

```

Csak a main rutinban történnek módosítások. A push, pop és a clear rutinok (K-R 85. oldal) változatlanok. A moduló operátor (%) használata hasonló az osztás (/) operátoréhoz. Az

```

op2 = pop();
if (op2 != 0.0)
    push((double) ((int)pop() % (int)op2));

```

moduló művelet a verem két felső elemén hajtódik végre. op2 a verem felső eleme, a program ezt hasonlítja össze a 0.0 értékkel mielőtt a moduló műveletet végrehajtaná. Minthogy a moduló művelet double típusú változókkal nem hajtható végre, a program a két operandust cast-okon keresztül egészekké konvertálja (K-R 53. oldal). A cast művelet eredménye double típusú mennyiséggé konvertálódik – szintén cast révén.

Az unáris mínusz operátor (m) leemeli a verem felső elemét, megváltoztatja az előjelét, majd visszahelyezi a verembe:

```

push(-pop());

```

A törölő parancs (e) meghívja az erase rutint. Az erase eggyel csökkenti a verem mutatóját és hibaüzenetet eredményez, ha a verem üres.

Egy változó a 26 nagybetű (A-Z) bármelyike lehet. A storage nevű double tömb 26 double szó helyét foglalja le. Az A változó tartalma a storage[0]-ban, a Z változóé a storage[25]-ben található. Mivel a storage external változónak van deklarálva (K-R 48. oldal), alapértelmezés szerint minden elemének kezdeti értéke 0.0.

Ha a program a bemeneten az (A-Z) változónevek valamelyikét észleli, a storage megfelelő eleme tolódik a veremre:

```

if (type >= 'A' && type <= 'Z')
    push(storage[type - 'A']);

```

Az (A-Z) változók mindegyikén tetszőleges művelet hajtható végre. Ha például A értéke 5, akkor a

```

6 A + =

```



összeadó művelet 6-ot ad A-hoz, aminek az eredménye 11. A változókon a tároló (s) művelet is végrehajtható. Ez a művelet a verem tetejét követő elemet rendeli hozzá a változóhoz:

```
pop();
storage[var - 'A'] = pop();
```

Ha például az 5 értéket akarjuk a B változóhoz rendelni, a parancs a következő:

```
5 B s
```

A tároló parancs (s) hatására kilép a B, kilép az 5, majd a storage[1]-ben (B-ben) tárolódik az 5.

A var változó a megelőző operandus/operátor típusával egyenlő. Minden alkalommal újra állítódik, amikor egy operandus/operátor érkezik:

```
var = type;
```

A var feladata, hogy eldöntse, vajon az utoljára érkezett elem változónév (A-Z) volt-e. A tároló parancs ezt az információt arra használja, hogy meghatározza, megadtak-e változónévet:

```
if (var >= 'A' && var <= 'Z')
```

Ha nem volt változónév megadva, hibaüzenet íródik ki.

#### 4.4. Gyakorlat: (K-R 90. oldal)

Írjuk meg az ungets(s) nevű rutint, amely egy teljes karakterláncot ír vissza a bemenetre! Szükséges, hogy az ungets függvénynek tudomása legyen buf-ról és bufp-ról, vagy csak egyszerűen használja az ungetch függvényt?

```
ungets(s)    /* karakterlanc visszahelyezese
               a bemenetre */
char s[];
{
    int len = strlen(s);          /* az s lanc hossza */
    while (len > 0)
        ungetch(s[--len]);
}
```

A len változó tartalmazza az s karakterláncban lévő karakterek számát (a záró '\0'-tól eltekintve). Ezt a számot a strlen függvény (K-R 48. oldal) határozza meg.

Az ungetch rutin (K-R 90. oldal) len-szer hívódik meg, minden hívás hatására egy karaktert az s karakterláncból visszatol a bemenetre. s utolsó karaktere (s[len-1])

tolódik vissza elsőként és s első karaktere (s[0]) tolódik vissza utoljára.

Az ungetch rutin használata révén az ungets leegyszerűsödik. ungets-nek nem kell ismernie a buf karaktertömböt, sem pedig a bufp egész típusú változót. Az ungetch rutin lekezeli buf-ot és bufp-t valamint elvégzi a hibavizsgálatot.

#### 4.5. Gyakorlat: (K-R 90. oldal)

Tegyük fel, hogy sohasem helyezünk vissza egynél több karaktert. Módosítsuk a getch és ungetch függvényeket ennek megfelelően!

```
#include <stdio.h>

int buf;          /* puffer az ungetch számára          */
                  /* EOF portabilis kezelése          */
int bufp = 0;     /* 0 ha a puffer üres, 1 ha foglalt */

getch()           /* beolvas egy (feltehetően visszahelyezett)
                  /* karaktert */
{
    if (bufp == 1) {
        bufp = 0;      /* üres puffer beállítása          */
        return(buf);
    } else
        return(getchar());
}

ungetch(c)        /* karakter visszahelyezése a bemenetre */
int c;
{
    if (bufp == 0) {
        bufp = 1;
        buf = c;
    } else
        printf("ungetch: túl sok karakter\n");
}
```

A buf nevű puffert most nem tömbnek deklaráljuk, mivel soha nem lesz egy karakternél több a pufferben. buf-t egésznek és nem karakternek deklaráljuk, hogy ezáltal gépfüggetlenül kezelhessük az EOF visszahelyezését az állományba (4.6. Gyakorlat).

A bufp egész típusú változó azt határozza meg, hogy a puffer üres-e, vagy van benne karakter. bufp értéke 0, ha a puffer üres és bufp egyenlő 1-gyel, ha foglalt.

A getch rutin ellenőrzi, hogy bufp egyenlő-e 1-gyel. Ha bufp értéke 1, akkor a rutin bufp-t 0-val teszi egyenlővé és a puffer tartalmát visszaadja. Egyéb esetben a getchar által beolvasott értéket adja vissza a hívó rutinnak.



Az `ungetch` rutin azt ellenőrzi, hogy `bufp` egyenlő-e 0-val. Ha `bufp` értéke 0, akkor a rutin `bufp`-t 1-gyel teszi egyenlővé és `c`-t a pufferba helyezi. Egyéb esetben hiba-üzenetet küld.

#### 4.6. Gyakorlat: (K-R 90. oldal)

A `getch` és `ungetch` rutinjaink a visszahelyezett EOF-ot gépfüggő módon kezelik. Határozzuk meg, hogyan viselkedjenek a rutinjaink, amikor EOF-ot írunk vissza, majd valósítsuk meg ezt a megoldást.

```
#include <stdio.h>
#define BUFSIZE 100

int buf[BUFSIZE];          /* puffer az ungetch számára */
                           /* EOF portabilis kezelése */
int bufp = 0;             /* a következő szabad hely buf-ban */

getch() /* beolvas egy (feltehetően visszahelyezett)
        karaktert */
{
    return((bufp > 0) ? buf[--bufp] : getchar());
}

ungetch(c) /* karakter visszahelyezése a bemenetre */
int c;
{
    if (bufp > BUFSIZE)
        printf("ungetch: túl sok karakter\n");
    else
        buf[bufp++] = c;
}
```

Feltehető, hogy az `stdio.h` fejinformációs állományban (header file-ban) az EOF szimbólikus állandó -1-nek van definiálva:

```
#define EOF (-1)
```

A `getch` és `ungetch` rutinokban (K-R 90. oldal) a `buf` puffer karaktertömbként van deklarálva:

```
char buf[BUFSIZE];
```

A C programozási nyelv nem írja elő, hogy egy `char` változó előjeles vagy előjel nélküli legyen (K-R 51. oldal). Amikor egy `char` `int`-té konvertálódik, előfordulhat, hogy sohasem kapunk negatív számot. Egyes számítógépeken, ha a `char` bal szélső bitje 1, az `int`-té való konverzió negatív számot eredményez. Más gépeken a konverzió úgy történik, hogy a `char` változó `int`-té való átalakítása során a bal oldalra nullák kerülnek. Az ilyen konverzió mindig pozitív számot eredményez, füg-

getlenül attól, hogy eredetileg a bal szélső bit 1 volt-e vagy sem.

Hexadecimális jelölésmódban a -1 ábrázolása 0xFFFF. Ha a 0xFFFF értéket char-ban tároljuk, a ténylegesen tárolt szám 0xFF. Ha a 0xFF-et konvertáljuk int-té, előfordulhat, hogy az eredmény 0x00FF, vagyis 255, más gépeken pedig 0xFFFF, ami -1.

| negatív szám (-1) | -> karakter | -> egész     |
|-------------------|-------------|--------------|
| 0xFFFF            | 0xFF        | 0x00FF (255) |
| 0xFFFF            | 0xFF        | 0xFFFF (-1)  |

Ha biztosítani akarjuk a programunk gépfüggetlenségét (portabilitását), akkor a buf puffert egészek tömbjeként kell deklarálnunk:

```
int buf[BUFSIZE];
```

Ekkor nem fordul elő konverzió és az EOF (-1) vagy bármely negatív szám kezelése gépfüggetlen módon történik.

#### 4.7. Gyakorlat: (K-R 96. oldal)

A printf-ben alkalmazott megoldások felhasználásával írjuk meg az itoa rekurzív változatát, vagyis rekurzív rutin segítségével konvertáljunk egy egész számot karakterlánccá!

```
itoa(n, s)          /* n karakterre konvertalasa s-be
                      (rekurziv változat) */
char s[];
int n;
{
    int sign;

    sign = (n < 0) ? -1 : 1;
    n = (n < 0) ? -n : n;
    itoar(n, s, 0, sign);
}

itoar(n, s, j, sign)
char s[];
int j, n, sign;
{
    int i;

    if ((i = n/10) != 0) {
        s[j++] = n % 10 + '0';
        itoar(i, s, j, sign);
    }

    if (i == 0) {
        s[j++] = n % 10 + '0';
        if (sign < 0)
            s[j++] = '-';
    }
}
```



```

    s[j] = '\0';
    reverse(s);
}
}

```

A megvalósítástól függetlenül meg kell tartanunk az itoa rutin korábbi felhasználói felületét (hívási és visszatérési módját). Ennélfogva csak a szám és a karakterlánc adódik át itoa-nak.

Ha csak azokat a nyelvi szerkezeteket használjuk fel, amelyek eddig a gyakorlatig már ismertnek tekinthetők, a megoldás meglehetősen nehézkes. itoa meghatározza a szám előjelét, majd meghívja az itoar rutint, amely végrehajtja a konverziót.

A rekurzív itoar rutin négy paramétert kap.  $n$  a konvertálandó szám. A jobb szélső számjegytől kezdve a konverzió számjegyenként történik.  $s$  a karakterlánc és  $j$  a pillanatnyi pozíció a láncban. Kezdetben  $j$  értéke 0. Az itoa által meghatározott sign (előjel) tartalma -1, ha a konvertálandó szám negatív és 1, ha pozitív.

Az itoar rutin két if utasítást tartalmaz.

Az első if utasítás, vagyis

```
if ((i = n/10) != 0)
```

leválasztja  $n$  jobb szélső számjegyét és az így fennmaradó számot  $i$ -ben tárolja. Például

```

n = 120   n/10 = 120/10 = 12   leválasztja a 0-t   i = 12
n =    9   n/10 =    9/10 =  0   leválasztja a 9-t   i =  0

```

Ha  $i$  nem 0, (vagyis maradt még számjegy), akkor az első if utasítás törzse hajtódik végre.  $n$  jobb szélső számjegye az ASCII megfelelőjére konvertálódik, a program eltárolja az  $s$  karakterláncba,  $j$  eggyel növekszik és az itoar rutin  $i$ -vel, mint konvertálandó számmal hívódik meg.

Ha  $i$  nulla, akkor a második if utasítás törzse hajtódik végre. Az  $n$ , ami végül csak egyetlen számjegyet tartalmaz, az ASCII megfelelőjére konvertálódik és  $s$ -ben tárolódik. Ha a sign (előjel) negatív, akkor azt a program  $s$ -hez illeszti, majd az  $s$  karakterláncot megfordítja (reverse( $s$ ), K-R 70. oldal). Ezekután  $s[0]$  vagy  $n$  bal szélső számjegyét, vagypedig a negatív előjelet tartalmazza.

Az unáris & operátort, amely egy objektum címét állítja elő, a K-R könyv a 99. oldalon említi először. Ennek ismeretében viszont az alábbi megoldást adhatjuk meg:

```

itoa(n, s)          /* n karakterre konvertalasa s-be
                      (rekurziv valtozat) */
char s[];
int n;
{
    int j, power;
    int i = 0;

    if (n < 0) {
        s[i++] = '-';          /* elojel          */
        n = -n;                /* n pozitivva valik */
    }
    power = 1;
    for (j = n; j / 10 != 0; j /= 10)      /* szamjegy */
        power *= 10;                      /* generalas helyes sorrendben */
    s[i++] = j + '0';

    if (n / 10 != 0)
        itoa(n-j*power, &s[i]);          /* kovetkezo szamjegy */
    else
        s[i] = '\0';
}

```

#### 4.8. Gyakorlat: (K-R 96. oldal)

Írjuk meg az s karakterláncot megfordító reverse(s) függvény rekurzív változatát!

```

reverse(s)          /* az s lanc megforditasa helyben
                      (rekurziv valtozat) */
char s[];
{
    reverser(s, 0, strlen(s));
}

reverser(s, i, len)
char s[];
int i;                /* bal oldali mutato a lancra */
int len;              /* a lanc hossza          */
{
    int c, j;

    j = len - (i + 1); /* jobb oldali mutato a lancra */

    if (i < j) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
        reverser(s, ++i, len);
    }
}

```

A megvalósítástól függetlenül a reverse felhasználói felületét (hívási és visszatérési módját) meg kell tartanunk. Ennélfogva a reverse-nek csak a karakterlánc adható át.



Ha csak azokat a nyelvi szerkezeteket használjuk fel, amelyek eddig a gyakorlatig már ismertnek tekinthetők, a megoldás meglehetősen nehézkes. reverse meghatározza a karakterlánc hosszát, majd meghívja a reverser függvényt, amely a lánc megfordítását végrehajtja.

A reverser függvény három bemeneti paramétere: s a megfordítandó karakterlánc, i a karakterlánc bal oldali mutatója és len a karakterlánc hossza (strlen(s), K-R 48. oldal). i kezdeti értéke 0. j a karakterlánc jobb oldali mutatója. j kiszámítása:

```
j = len - (i + 1);
```

A láncon belüli karakterek felcserélése kívülről befelé történik. Például az első két, egymással felcserélt karakter s[0] és s[len-1], a másodikként felcserélt karakterek s[1] és s[len-2]. Minden alkalommal, amikor a reverser meghívódik, az i bal oldali mutató eggyel növekszik:

```
reverser(s, ++i, len);
```

A felcserélés mindaddig folytatódik, amíg vagy mindkét mutató ugyanarra a karakterre nem mutat (i==j), vagy pedig a bal oldali mutató a jobb oldali mutatótól jobbra eső karakterre nem mutat (i>j).

A mutatókról a K-R könyv 99. oldaláig nem esik szó. Ezek ismeretében viszont az alábbi megoldást adhatjuk:

```
reverse(s)          /* az s lanc megforditasa helyben
                      (rekurziv változat) */
char s[];
{
    reverser(s, s + strlen(s) - 1);
}

reverser(s, t)
char *s;              /* bal oldali mutato a lancra */
char *t;              /* jobb oldali mutato a lancra */
{
    int c;

    if (s < t) {
        c = *s;
        *s = *t;
        *t = c;
        reverser(++s, --t);
    }
}
```

A megoldás még mindig nehézkes, de az első megoldásnál jobb, mivel eggyel kevesebb argumentumot ad át reverser-nek és indexelés helyett mutatókat használ.

Kezdetben s a karakterlánc első karakterére mutat, míg t

a lánc utolsó karakterére (nem számítva a lánczáró '\0'-t).

reverser felépítése hasonló az első megoldásban látott-hoz.

#### 4.9. Gyakorlat: (K-R 98. oldal)

Definiáljuk a swap(x,y) makrót, amely megcseréli a két int típusú argumentumát! (A blokkstruktúra segítségünkre lesz.)

```
#define swap(x, y) { int _z; _z = y; y = x; x = _z; }
```

A swap makró akkor működik helyesen, ha az argumentumok egyike sem \_z. Ha az argumentumok valamelyikének neve \_z, akkor a makró kiterjesztésének eredménye

```
{ int _z; _z = _z; _z = x; x = _z; }
```

és az eredmény definiálatlan. A megoldás feltételezi tehát, hogy változónévként az \_z nem fordul elő.

```
#define swap(x, y) { x ^= y; y ^= x; x ^= y; }
```

Ez a megoldás a bitenkénti kizáró VAGY műveletre (^) épül. Az alábbi táblázatban az előállított z értékeket tüntettük fel x és y különféle értékei esetén:

| x | ^ | y | = | z |
|---|---|---|---|---|
| - |   | - |   | - |
| 0 |   | 0 |   | 0 |
| 0 |   | 1 |   | 1 |
| 1 |   | 0 |   | 1 |
| 1 |   | 1 |   | 0 |

x és y helycseréje az x és y között háromszor végrehajtott kizáró VAGY művelet eredményeként áll elő:

```
x = x ^ y;  
y = x ^ y;  
x = x ^ y;
```

Az első kizáró VAGY művelet x-et a maszkkal teszi egyenlővé. A maszk egy adott bitje akkor lesz 1, ha az eredeti x-ben és y-ban a megfelelő bitek különböznek és akkor lesz 0, ha az eredeti x-ben és y-ban a megfelelő bitek azonosak. A második kizáró VAGY művelet eredményeképpen y az eredeti x-szel lesz egyenlő, mégpedig a maszk-ból és az eredeti y-ból vett információ alapján. A harmadik kizáró VAGY művelet hatására x az eredeti y-nal lesz egyenlő, mégpedig a maszk-ból és az új y-ból (az eredeti x-ből) vett információ alapján.



## Mutatók és tömbök

### 5.1. Gyakorlat: (K-R 103. oldal)

Írjuk meg a `getfloat` függvényt, amely a `getint` lebegőpontos megfelelője! Milyen típust ad vissza a `getfloat` függvényértékként?

```
#include <stdio.h>

getfloat(pn)          /* kov. lebegopontos szam olvasasa
                        a bemenetrol */
float *pn;
{
    int c, sign;
    float power;

    while ((c=getch()) == ' ' || c == '\n' || c == '\t')
        ;          /* atlepi a lathatatlan karaktereket */
    sign = 1;
    if (c == '+' || c == '-') {
        /* feljegyzzi az elojelet */
        sign = (c == '+') ? 1 : -1;
        c = getch();
    }
    for (*pn = 0.0; c >= '0' && c <= '9'; c = getch())
        *pn = 10.0 * *pn + c - '0';          /* egesz resz */
    if (c == '.')
        c = getch();
    for (power = 1.0; c >= '0' && c <= '9'; c = getch()){
        *pn = 10.0 * *pn + c - '0';          /* tortresz */
        power *= 10.0;
    }
    *pn *= sign / power;          /* vegso szam */
    if (c != EOF)
        ungetch(c);
    return(c);
}
```

A `getfloat` rutin hasonló a `getint` rutinhoz (K-R 102. oldal). `getfloat` átlépi a láthatatlan közöket, feljegyzzi az előjelet és a `pn` mutató által megcímzett helyen tárolja a szám egész részét.

A `getfloat` ugyancsak lekezezi a szám tört részét (de a tudományos jelölésmódú számokat nem). A tört részt ugyanúgy adja `*pn`-hez, mint az egész részt:

```
*pn = 10.0 * *pn + c - '0';
```

A továbbiakban a program `*pn`-et elosztja a `power` nevű változóval, ezáltal elvégzi a számnak a tört rész jelen-

léte következtében szükséges módosítását. A tizedespon-  
tot követő minden számjegy beolvasása után a program a  
power változót tizzel megszorozza. Ha például a tizedes-  
pontot nem követi számjegy, a power értéke 1, ha egy  
számjegy követi, akkor power értéke 10, ha három szám-  
jegy követi, akkor pedig power értéke 1000 lesz.

Miután a program a törtrész összes számjegyét beolvasta,  
kiszámítja a szám végső értékét. Ezt az értéket megszo-  
rozza a sign-nal és elosztja a power-rel:

```
*pn *= sign / power;
```

getint-hez hasonlóan, a visszatérő érték vagy EOF, vagy  
pedig a számot követő karakter ASCII értéke. A visszatér-  
ő érték típusa szintén int.

## 5.2. Gyakorlat: (K-R 111. oldal)

Írjuk át a 2. fejezetben bemutatott strcat függvényt  
mutató alkalmazásával [strcat(s, t) a t karakterláncot  
az s karakterlánc végére másolja!]

```
strcat(s, t)          /* t illesztése s vegere          */
char *s, *t;          /* mutatókkal működő változat    */
{
    while (*s)
        s++;
    while (*s++ = *t++)
        ;
}
```

s és t kezdetben a karakterláncok elejére mutat.

Az első while ciklus addig növeli az s mutatót, amíg meg  
nem találja a lánc végét jelző '\0'-t. A

```
while (*s)
```

utasítás mindaddig igaz értékű, amíg a program meg nem  
találja a lánczáró karaktert (\*s != '\0').

A második while ciklus az s karakterlánchoz hozzáfűg-  
geszti a t karakterláncot:

```
while (*s++ = *t++)
    ;
```

A fenti utasítás egyenlővé teszi az s által megcímzett  
objektumot azzal, amire t mutat (\*s = \*t), mindkét  
mutatót eggyel megnöveli (s++, t++), majd ugyanezt foly-  
tatja mindaddig, amíg t a lánc végét jelző karakterre  
nem mutat (\*t != '\0').



### 5.3. Gyakorlat: (K-R 111. oldal)

Írjunk makrót strcpy-ra!

```
#define strcpy(s, t)      { char *__ps = s, *__pt = t; \
                           while (*__ps++ = *__pt++) \
                               ; \
                           }
```

Az strcpy makró csak akkor működik helyesen, ha egyik argumentuma sem \_ps vagy \_pt. A makró működése hasonló az strcpy rutinéhoz (K-R 110. oldal).

Mivel a makródefiníció hosszú, a következő sorban való folytatás érdekében a folytatandó sor végén egy \ karaktert helyezünk el (K-R 97. oldal).

### 5.4. Gyakorlat: (K-R 111. oldal)

Írjuk át a korábbi fejezetek erre alkalmas programjait és gyakorlatait úgy, hogy a tömbindexelés helyett mutatókat használunk! Jó lehetőség erre pl. a getline (1. az 1. és 4. fejezetet), az atoi és az itoa és változataik (1. a 2., 3. és 4. fejezetet), a reverse (3. fejezet) valamint az index és a getop (4. fejezet).

```
#include <stdio.h>

getline(s, lim)          /* sor beolvasása s-be */
char *s;                 /* visszaadja a hosszat */
int lim;
{
    int c;
    char *t = s;

    while (--lim > 0 && (c = getchar()) != EOF && c != '\n')
        *s++ = c;
    if (c == '\n')
        *s++ = c;
    *s = '\0';
    return(s - t);
}
```

Az s karaktertömböt a \*s karaktermutató helyettesíti. Most már nincs szükségünk az i változóra, amely korábban az s karaktertömb indexe volt. A karakterláncon történő indexelés az s mutató inkrementálásával történik.

Az

```
s[i++] = c;
```

utasítás egyenértékű a

```
*s++ = c;
```

utasítással. Amikor s átadódik getline-nak, a karakterlánc első elemére mutat. \*t szintén karaktermutató, a program úgy állítja be, hogy szintén erre az első elemre mutasson:

```
char *t = s;
```

Miután egy sor beérkezett, s a zárókarakterre ('\0') mutat. Mivel t még mindig a sor első karakterére mutat, így a sor hossza s - t.

```
atoi(s)          /* s konvertálása egéssze */
char *s;          /* mutatókkal működő változat */
{
    int n, sign;

    for ( ; *s == ' ' || *s == '\n' || *s == '\t'; s++)
        ; /* atugorja a láthatatlan karaktereket */
    sign = 1;
    if (*s == '+' || *s == '-') /* a szám előjele */
        sign = (*s++ == '+') ? 1 : -1;
    for (n = 0; *s >= '0' && *s <= '9'; s++)
        n = 10 * n + *s - '0'; /* a szám értéke */
    return(sign * n);
}
```

Az s[i] egyenértékű \*s-sel, az s[i++] pedig az \*s++ kifejezéssel.

```
itoa(n, s)        /* n konvertálása s-beli karakterekké */
char *s;          /* mutatókkal működő változat */
int n;
{
    int sign;
    char *t = s; /* s mentése (mutató a lánc elejére) */

    if ((sign = n) < 0) /* feljegyzi az előjelet */
        n = -n;
    do { /* számjegyek előállítása fordított
                                                sorrendben */
        *s++ = n % 10 + '0'; /* veszi a következő számjegyet */
    } while ((n /= 10) > 0); /* törli */
    if (sign < 0)
        *s++ = '-';
    *s = '\0';
    reverse(t);
}
```

A program a \*t karaktermutatót úgy inicializálja, hogy a karakterlánc első elemére mutasson:

```
char *t = s;
```



A

```
*s++ = n % 10 + '\0';
```

utasítás egyenértékű az

```
s[i++] = n % 10 + '\0';
```

utasítással.

A reverse rutinnak a t mutató adódik át, amely továbbra is a lánc első elemére mutat. s nem adódik át, mivel a karakterlánc végére mutat (a '\0' nulla zárókarakterre).

```
#include <stdio.h>
```

```
reverse(s)          /* az s lánc megfordítása helyben */  
char *s;             /* mutatókkal működő változat */  
{  
    char *t;  
    int c;  
  
    for (t = s + (strlen(s)-1); s < t; s++, t--) {  
        c = *s;  
        *s = *t;  
        *t = c;  
    }  
}
```

Az s karaktertömböt a \*s karaktermutató váltja fel. Az i és j változókra, amelyeket az s tömb indexelésére használtunk, a továbbiakban nincs szükség.

Kezdetben s a lánc első elemére mutat. A \*t karaktermutató pedig kezdetben a lánc utolsó elemére (a záró '\0'-t megelőző elemre) mutat:

```
t = s + (strlen(s) - 1)
```

\*s egyenértékű s[i]-vel és \*t egyenértékű t[j]-vel. A for cikluson belüli ellenőrzés, vagyis

```
s < t
```

ekvivalens az

```
i < j
```

ellenőrzéssel. s++ ugyanazt eredményezi, mint az i index inkrementálása (i++) és t-- ugyanazt eredményezi, mint a j index dekrementálása (j--).

```
index(s, t)          /* visszaadja t s-beli indexet */  
char *s, *t;          /* mutatókkal működő változat */  
{  
    char *b = s;        /* az s lánc kezdete */
```

```

char *p, *r;

for ( ; *s != '\0'; s++) {
    for(p=s, r=t; *r != '\0' && *p == *r; p++, r++)
        ;
    if (*r == '\0')
        return(s - b);
}
return(-1);
}

```

s[i]-t \*s, s[j]-t \*p és t[k]-t \*r helyettesíti. \*b karaktermutató, amely mindig az s karakterlánc első elemére mutat (s[0]). p = s egyenértékű j = i-vel. r = t pedig k = 0-val.

Amennyiben az if utasítás igaz, vagyis

```
if (*r == '\0')
```

akkor a program egyezést talált és a t s láncbeli indexét adja vissza:

```
return(s - b);
```

```

double atof(s) /* az s lanc konvertalasa double-lá */
char *s; /* mutatókkal működő változat */
{
    double val, power;
    int sign;

    for ( ; *s == ' ' || *s == '\n' || *s == '\t'; s++)
        ; /* atugorja a láthatatlan karaktereket */
    sign = 1;
    if (*s == '+' || *s == '-') /* a szám előjele */
        sign = (*s++ == '+') ? 1 : -1;
    for (val = 0; *s >= '0' && *s <= '9'; s++)
        val = 10 * val + *s - '0';
    if (*s == '.')
        s++;
    for (power = 1; *s >= '0' && *s <= '9'; s++) {
        val = 10 * val + *s - '0';
        power *= 10;
    }
    return(sign * val / power);
}

```

s[++] egyenértékű \*s++ -szal.

```

#include <stdio.h>
#define NUMBER '0' /* jelzi, hogy számot talált */
#define TOOBIG '9' /* jelzi, hogy a lanc túl hosszú */

getop(s, lim) /* veszi a következő operátort vagy
               operandust */

```



```

char *s;          /* mutatokkal mukodo változat */
int lim;
{
    char *sl = s + lim;
    int c;

    while ((c=getch()) == ' ' || c == '\t' || c == '\n')
        ;
    if (c != '.' && (c < '0' || c > '9'))
        return(c);
    *s = c;
    for (s++; (c = getchar()) >= '0' && c <= '9'; s++)
        if (s < sl)
            *s = c;
    if (c == '.') {
        /* veszi a törtreszt */
        if (s < sl)
            *s = c;
        for (s++; (c=getchar()) >= '0' && c <= '9'; s++)
            if (s < sl)
                *s = c;
    }
    if (s < sl) {
        /* a szám rendben van */
        ungetch(c);
        *s = '\0';
        return(NUMBER);
    } else {
        /* túl nagy; atugorja a sor maradékát */
        while (c != '\n' && c != EOF)
            c = getchar();
        *(sl-1) = '\0';
        return(TOOBIG);
    }
}
}

```

Az `s` tömb két indexét összehasonlító `if` utasítást, vagyis az

```
if (i < lim)
```

utasítást az

```
if (s < sl)
```

utasítással helyettesítettük. Ez az utasítás két címet hasonlít össze. Mivel `s` és `sl` ugyanarra a tömbre mutatnak, az ellenőrzés azt dönti el, hogy `s` a tömbnek egy korábbi elemére mutat-e, mint `sl`.

### 5.5. Gyakorlat: (K-R 117. oldal)

Írjuk újra a `readlines` függvényt oly módon, hogy a sorokat a `main` által adott tömbben hozzuk létre és nem az `alloc`-ot hívjuk a tár karbantartása céljából! Mennyivel gyorsabb így a program?

```

#define MAXLEN 100    /* maximalis sorhossz */
#define MAXSTOR 1000 /* az adott tarterulet merete */

readlines(lineptr, linestor, maxlines) /* bemeneti */
char *lineptr[]; /* sor olvasasa rendezeshez */
char linestor[];
int maxlines;
{
    int len, nlines;
    char line[MAXLEN];
    char *p = linestor;
    char *linestop = linestor + MAXSTOR;

    nlines = 0;
    while ((len = getline(line, MAXLEN)) > 0)
        if (nlines >= maxlines)
            return(-1);
        else if (p + len > linestop)
            return(-1);
        else {
            line[len-1] = '\0';
            strcpy(p, line);
            lineptr[nlines++] = p;
            p += len;
        }
    return(nlines);
}

```

A main rutin rendelkezésre bocsátja a linestor tömböt, amely a readlines által behozott sorokat tárolja. A \*p karaktermutatót a program úgy inicializálja, hogy az a linestor első elemeire mutasson:

```
char *p = linestor;
```

Az eredeti readlines rutin (K-R 115. oldal) az alloc rutint használja (K-R 105. oldal): az

```
else if (p + len > linestop)
```

utasítás garantálja, hogy a linestor-ban van még hely.

A readlines átírása a programot valamivel meggyorsítja.

## 5.6. Gyakorlat: (K-R 118. oldal)

Írjuk át a day\_of\_year és a month\_day rutint oly módon, hogy indexelés helyett mutatókat használjunk!

```

static int a[13] = {
    0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
};
static int b[13] = {
    0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
};

```



```

static int *day_tab[2] = {a, b};

day_of_year(year, month, day) /* nap sorszáma az év */
int year, month, day; /* hónapja és napja szerint */
{
    int leap, *p;

    leap=year%4 == 0 && year%100 != 0 || year%400 == 0;

    p = *(day_tab + leap);
    while (--month)
        day += *++p;
    return(day);
}

month_day(year, yearday, pmonth, pday)
/* hónap és nap a nap évbeli sorszámból */
int year, yearday, *pmonth, *pday;
{
    int leap, *p;

    leap=year%4 == 0 && year%100 != 0 || year%400 == 0;

    p = *(day_tab + leap);
    while (yearday > *++p)
        yearday -= *p;
    *pmonth = p - *(day_tab + leap);
    *pday = yearday;
}

```

Az eredeti, kétdimenziós, egész típusú day\_tab tömbből most két különálló tömböt (a, b) csinálunk, amelyek mindegyike 13 elemű. \*day\_tab mutatók tömbje. \*day\_tab első eleme (\*day\_tab[0]) az a[0]-ra mutat, \*day\_tab második eleme (\*day\_tab[1]) b[0]-ra mutat.

A p mutató kezdetben vagy a[0]-ra (leap értéke 0), vagy b[0]-ra mutat (leap értéke 1):

```
p = *(day_tab + leap);
```

Az eredeti day\_of\_year rutin for ciklusa, vagyis

```
for (i = 1; i < month; i++)
    day += day_tab[leap][i];
```

egyenértékű a

```
p = *(day_tab + leap);
while (--month)
    day += *++p;
```

utasításokkal a módosított day\_of\_year rutinban.

Az eredeti month\_day rutinbeli

```
for(i = 1; yearday > day_tab[leap][i]; i++)
    yearday -= day_tab[leap][i];
```

egyenértékű a

```
p = *(day_tab + leap);
while (yearday > *++p)
    yearday -= *p;
```

utasításokkal a módosított month\_day rutinban.

\*p vagy egy a vagy egy b-beli elem. \*++p először eggyel megnöveli a mutatót, utána nyúl az elemhez.

Mivel a program nem használ indexelést, \*pmonth kiszámítása a

```
*pmonth = p - *(day_tab + leap);
```

utasítással történik.

## 5.7. Gyakorlat: (K-R 123. oldal)

Írjuk meg az add nevű programot, amely kiértékel egy, a parancssorban előforduló fordított lengyel alakú kifejezést! Például

```
add 2 3 4 + *
```

2 x (3+4)-et számítja ki.

```
#include <stdio.h>
#define MAXOP 20 /* operandus, operator maximalis merete */
#define NUMBER '0' /* jelzi, hogy számot talált */
#define UNKNOWN '8' /* ismeretlen operandus/operator */
#define TOOBIG '9' /* túl hosszú lanc jelzése */

main(argc, argv) /* fordított lengyel kalkulator */
int argc; /* parancssor változat */
char *argv[];
{
    int type;
    char s[MAXOP];
    double op2, atof(), pop(), push();

    while (--argc)
        switch (type = gettype(s, MAXOP, *++argv)) {
            case NUMBER:
                push(atof(s));
                break;
            case '+':
                push(pop() + pop());
                break;
            case '*':
                push(pop() * pop());
                break;
            case '-':
```



```

        op2 = pop();
        push(pop() - op2);
        break;
    case '/':
        op2 = pop();
        if (op2 != 0.0)
            push(pop() / op2);
        else
            printf("nulla osztó leemelve\n");
        break;
    case '=':
        printf("\t%f\n", push(pop()));
        break;
    case 'c':
        clear();
        break;
    case TOOBIG:
        printf("%.20s ...tul hosszú\n", s);
        argc = 1;
        break;
    case UNKNOWN:
    default:
        printf("%.20s ...ismeretlen operandus/operator\n", s);
        argc = 1;
        break;
    }
}

gettype(s, lim, t)      /* parancssor argumentum típus
                           meghatározása */
char *s, *t;
int lim;
{
    char *s1 = s + lim;

    if (*t != '.' && (*t < '0' || *t > '9'))
        if (++t) { /* ismeretlen operator */
            *s++ = *(t-1);
            *s++ = *t;
            *s = '\0';
            return(UNKNOWN);
        } else
            return((int)*(t-1)); /* operator */
    *s++ = *t; /* egész rész */
    while (++t >= '0' && *t <= '9')
        if (s < s1)
            *s++ = *t;
    if (*t == '.') { /* törtész */
        if (s < s1)
            *s++ = *t;
        while (++t >= '0' && *t <= '9')
            if (s < s1)
                *s++ = *t;
    }
    if (*t) { /* ismeretlen operandus */
        *s++ = *t;
        *s = '\0';
        return(UNKNOWN);
    }
}

```

```

if (s < sl) { /* a szám rendben van */
    *s = '\0';
    return(NUMBER);
} else { /* a szám túl nagy */
    *(sl-1) = '\0';
    return(TOOBIG);
}
}

```

A megoldás a K-R könyv 84. oldalán közölt "reverse Polish" (fordított lengyel elvű) kalkulátorprogramon alapul. Felhasználja a push, pop és clear rutinokat (K-R 85. oldal).

A parancssorból beolvasott minden operátort és operandust legalább egy üres köznék hely elválasztania a szomszédjától. A gettype azt dönti el, hogy az argumentum ismert típusú-e. Ha az argumentum szám vagy a (+, -, \*, /, =, c) operátorok valamelyike, akkor a megfelelő értéket adja vissza. Ha az argumentum valami más, akkor vagy az UNKNOWN (ismeretlen) vagy a helytelen értéket adja vissza.

A gettype rutin öt részre osztható.

Az első if utasítás, vagyis

```
if (*t != '.' && (*t < '0' || *t > '9'))
```

igaz, ha az argumentum első eleme nem számjegy és nem tizedespont. Az

```
if (*++t)
```

feltétel csak akkor hamis, ha az argumentum egy karaktert és a záró '\0'-t tartalmazza. A rutin ennek a karakternek az értéket adja vissza. Ha a feltétel hamis, akkor a rutin ismeretlen operátort talál.

Ha az argumentum szám, a következő utasításcsoport, vagyis

```

*s++ = *t;
while (*++t >= '\0' && *t <= '9')
    if (s < sl)
        *s++ = *t;

```

mindaddig beolvassa és \*s-ben eltárolja az argumentumot, amíg van elegendő tárolóhely (s < sl).

Következőként a program a tizedespontot és a törtrészt olvassa be. Ezt a részt nem hajtja végre, ha nincs tizedespont (\*t != '.').

Ha a '\0'-t kivéve a törtrészt valami más is követi, vagy a tizedesponton, illetve '\0'-n kívül valami más is van az egész rész után, akkor az operandus ismeretlen típusú és a visszaadott érték az UNKNOWN lesz.



Végezetül, ha a szám eltárolására elegendő tárolóhely áll rendelkezésre, akkor a NUMBER tér vissza. Ha viszont nincs elegendő hely, akkor a TOO BIG (túl nagy) adódik vissza.

Ha az argumentumok beolvasása után hiba fordul elő, argc 1-re állítódik. A main rutin beli while ciklus, azaz

```
while (--argc)
```

hamissá válik és a program véget ér.

### 5.8. Gyakorlat: (K-R 123. oldal)

Módosítsuk az entab és detab programokat (amelyeket az 1. fejezetben gyakorlatként írtunk meg) oly módon, hogy az a tabulátor stop-ok listáját argumentumokként fogadja! Argumentum megadásának hiányában a normál tabulátor-beállításokat használjuk!

```
#include <stdio.h>
#define MAXLINE 100      /* maximalis sormeret */
#define TABINC 8         /* alapert. tabulator
                           inkremens meret */
#define YES 1
#define NO 0

main(argc, argv)        /* entab: szokozlanc
                           helyettesitese */
int argc;               /* tab-okkal es szokozokkal; */
char *argv[];           /* parancssor valtozat */
{
    int tabl[MAXLINE+1];

    settab(argc, argv, tab); /* tabulatorstopok
                               inicializalasa */
    entab(tab);             /* szokozlanc helyettesitese */
                             /* tab-okkal es szokozokkal */
}

settab(argc, argv, tab) /* tabulatorstopok
                           inicializalasa */
int argc;
char *argv[];
int tab[];
{
    int i, pos;

    if (argc <= 1) /* alapertelmezett tab-stopok */
        for (i = 1; i <= MAXLINE; i++)
            if (i % TABINC == 1)
                tab[i] = YES;
            else
                tab[i] = NO;
    else { /* megadott tab-stopok */
```

```

    for (i = 1; i <= MAXLINE; i++)
        tab[i] = NO; /* minden tab-stopot kikapcsol */
    while (--argc) { /* vegigmegy az arg listan */
        pos = atoi(++argv);
        if (pos > 0 && pos <= MAXLINE)
            tab[pos] = YES; /* bekapcsolja */
    } /* a kiválasztottakat */
}

entab(tab) /* szokozlanc helyettesítése */
int tab[]; /* tab-okkal és szokozokkal */
{
    int c, pos;
    int nb = 0; /* a szükséges szokozok száma */
    int nt = 0; /* a szükséges tab-ok száma */

    for (pos = 1; (c=getchar()) != EOF; pos++)
        if (c == ' ')
            if (tabpos(pos+1, tab) == NO)
                ++nb; /* szokoz behelyezése */
            else {
                nb = 0; /* szokozok helyett tab */
                ++nt;
            }
        else {
            while (nt) { /* kimenet */
                putchar('\t');
                --nt;
            }
            if (c != '\t')
                while (nb > 0) {
                    putchar(' ');
                    --nb;
                }
            else
                nb = 0;
            putchar(c);
            if (c == '\n')
                pos = 0;
            else if (c == '\t')
                while (tabpos(pos+1, tab) != YES)
                    ++pos;
        }
    }

    tabpos(pos, tab) /* eldönti, hogy a pozíció van-e
                                tabulatorstop */

    int pos;
    int tab[];
    {
        if (pos > MAXLINE)
            return(YES);
        else
            return(tab[pos]);
    }
}

```



A megoldás elve Kernighan & Plauger, Software Tools (Addison-Wesley, 1976) című könyvében közölt entab programmal azonos. A könyv magyar kiadásának címe: A programozás magasiskolája (Műszaki Könyvkiadó, 1982, lásd a 47.-48. oldalt).

A tab tömb minden eleme megfelel egy-egy sorbeli pozíciónak, azaz tab[i] az első sorbeli pozíciónak felel meg (pos egyenlő 1-gyel). Ha a sorbeli pozíción egy tabulátorstop van, akkor a megfelelő tab[i] elem értéke YES, egyébként tab[i] értéke NO.

A tabulátorstopok inicializálása a settab rutinban történik. Ha nincs argumentum (argc egyenlő 1-gyel), akkor a program abban az esetben talált tabulátor stopot, ha az

```
if (i % TABINC == 1)
```

feltétel igaz. Ha vannak argumentumok, akkor a parancssorból beolvasott egyes tabulátorstopoknak megfelelő tab-beli elemek értéke YES lesz.

Az entab rutin a szóközök sorozatait szóközökkel és tab-okkal helyettesíti. Ez a rutin hasonló az 1.20. Gyakorlatban közölthöz.

A tabpos rutin eldönti, hogy az adott pozíció tabulátorstop-e, illetve, hogy a pozíció nagyobb-e, mint a sorbeli karakterek maximális száma (MAXLINE). Ha a pozíció nagyobb, mint MAXLINE, akkor YES adódik vissza, egyébként pedig tab[pos].

```
#include <stdio.h>
#define MAXLINE 100      /* maximalis sormeret */
#define TABINC 8         /* alapert. tabulator
                           inkremens meret */
#define YES 1
#define NO 0

main(argc, argv)        /* detab: tab karakter
                           helyettesitese szokozokkal */
int argc;                /* parancssor valtozat */
char *argv[];
{
    int tab[MAXLINE+1];

    settab(argc, argv, tab); /* tabulatorstopok
                               inicializalasa */
    detab(tab);             /* tab helyettesitese szokozokkal */
}

detab(tab)               /* tab helyettesitese szokozokkal */
int tab[];
{
    int c, pos = 1;
```

```

while ((c = getchar()) != EOF)
    if (c == '\t') {                                /* tab */
        do
            putchar(' ');
            while (tabpos(++pos, tab) != YES);
        } else if (c == '\n') {                      /* ujsor */
            putchar(c);
            pos = 1;
        } else {                                     /* egyeb */
            putchar(c);
            ++pos;
        }
    }
}

```

A megoldás elve Kernighan & Plauger, Software Tools (A programozás magasiskolája, Addison-Wesley, 1976, ill. Műszaki Könyvkiadó, 1982, 32-33. old.) című könyvében közölt detab programmal azonos.

A tabpos és settab rutinok ugyanazok, mint amiket a gyakorlat első részében láttunk.

A detab rutin a tab-okat szóközzel helyettesíti. Ez hasonló az 1.19. Gyakorlatban látott rutinhoz. Ha a program egy tab karaktert talál, akkor mindaddig szóközt illeszt a láncba, amíg tabpos(++pos, tab) értéke YES nem lesz:

```

do
    putchar(' ');
while (tabpos(++pos, tab) != YES);

```

## 5.9. Gyakorlat: (K-R 123. oldal)

Bővítsük ki az entab és detab programokat úgy, hogy elfogadják az

entab m + n

rövidített jelölést, amely az m-edik oszloptól kezdve minden n-edik oszlopon egy-egy tabulátorstopot jelent! Válasszunk (a felhasználó számára) kényelmes alapértelmezést (default)!

```

#include <stdio.h>
#define MAXLINE 100    /* maximalis sormeret */
#define TABINC 8       /* alapert. tabulator
                        inkremens meret */
#define YES 1
#define NO 0

main(argc, argv)      /* entab: szokozlanc */
int argc;             /* helyettesitese tab-okkal es */
char *argv[];         /* szokozokkal; parancssor valtozat */

```



```

{
    int tab[MAXLINE+1];

    esettab(argc, argv, tab); /* tabulatorstopok
                                inicializalasa */
    entab(tab);               /* szokozlanc helyettesitese */
                                /* tab-okkal es szokozokkal */
}

esettab(argc, argv, tab)      /* tabulator stopok
                                inicializalasa */
int argc;                    /* kibovitett valtozat */
char *argv[];
int tab[];
{
    int i, inc, pos;

    if (argc <= 1) /* alapertelmezett tab-stopok */
        for (i = 1; i <= MAXLINE; i++)
            if (i % TABINC == 1)
                tab[i] = YES;
            else
                tab[i] = NO;
    else if (argc == 3 && *argv[2] == '+') { /* m + n */
        pos = atoi(++argv);
        inc = atoi(++argv);
        for (i = 1; i <= MAXLINE; i++)
            if (i == pos) {
                tab[i] = YES;
                pos += inc;
            } else
                tab[i] = NO;
    } else { /* megadott tabulatorstopok */
        for (i = 1; i <= MAXLINE; i++)
            tab[i] = NO; /* minden tab-stopot kikapcsol */
        while (--argc) { /* vegigmegy az arg listan */
            pos = atoi(++argv);
            if (pos > 0 && pos <= MAXLINE)
                tab[pos] = YES; /* a kivlasztottakat */
        } /* bekapcsolja */
    }
}
}

```

A megoldás elve Kernighan & Plauger, Software Tools (A programozás magasiskolája, Addison-Wesley, 1976, ill. Műszaki Könyvkiadó, 1982, 47-48. old.) című könyvében közölt entab programmal azonos.

A megoldás hasonló az 5.8 Gyakorlatban látott entab programhoz. Az egyetlen változtatás, hogy a settab rutint az esettab rutin (bővített settab) váltja fel.

Az esettab a rövidített "m+n" jelölésmódot fogadja. A

```

pos = atoi(++argv);
inc = atoi(++argv);

```

utasítások a pos-t az első tabulatorstop pozíciójával,

inc-et pedig az inkremensmérettel teszik egyenlővé. Ennek következtében a tabulátorstopok a pos-on kezdődnek és minden inc-edik pozíción fordulnak elő.

```
#include <stdio.h>
#define MAXLINE 100 /* maximalis sormeret */
#define TABINC 8 /* alapert. tabulator
                    inkremens meret */

#define YES 1
#define NO 0

main(argc, argv) /* detab: tab helyettesítése */
int argc; /* szokozokkal: parancssor */
char *argv[]; /* változat (m+n) */
{
    int tab[MAXLINE+1];

    esettab(argc, argv, tab); /* tabulatorstopok
                                inicializalasa */
    detab(tab); /* tab helyettesítése szokozokkal */
}
```

A megoldás kerete Kernighan & Plauger, Software Tools (A programozás magasiskolája, Addison-Wesley, 1976, ill. Műszaki Könyvkiadó, 1982, 32-33. old.) című könyvében közölt detab programmal azonos.

A megoldás hasonló az 5.8 Gyakorlatban látott entab programhoz és a gyakorlat első részében közölt esettab rutint használja.

#### 5.10. Gyakorlat: (K-R 123. oldal)

Írjuk meg a tail nevű programot, amely kinyomtatja az utolsó n bemeneti sort! Az n alapértelmezése legyen pl. 10, amit opcionális argumentum változtathat meg, tehát

```
tail -n
```

az utolsó n sort nyomtatja ki. A programnak mindig elfogadhatóan kell viselkednie, függetlenül attól, hogy mennyire értelmetlen a bemenet vagy n értéke. Írjuk meg úgy a programot, hogy a legjobban kihasználja a rendelkezésre álló tárterületet: a sorokat tároljuk úgy, mint a sort programban, nem pedig rögzített méretű kétdimenziós tömbben!

```
#include <stdio.h>
#define DEFLINES 10 /* ny. sorok szama - alapert. */
#define LINES 100 /* nyomt. sorok maximalis szama */
#define MAXLEN 100 /* egy bemeneti sor max. hossza */

main(argc, argv) /* tail: az utolso n bemeneti sor */
int argc; /* kinyomtatasa */
char *argv[];
```



```

{
    int first, i, last, len, n, nlines;
    char *alloc(), *p;
    char line[MAXLEN];          /* aktualis bemeneti sor */
    char *lineptr[LINES];       /* mutatok a beolvasott
                                   sorokra */

    if (argc == 1)              /* nincs argumentum megadva */
        n = DEFLINES;          /* alapertelm. darabszam */
    else if (argc == 2)
        if ((*++argv)[0] == '-')
            n = atoi(argv[0]+1);
        else
            error("hasznalat: tail [-n]");
    else
        error("hasznalat: tail [-n]");

    if (n < 1 || n > LINES)     /* n erteke ertelmetlen? */
        n = LINES;
    for (i = 0; i < LINES; i++)
        lineptr[i] = NULL;
    last = 0;                  /* az utoljara beolvasott sor indexe */
    nlines = 0;                /* a beolvasott sorok szama */

    while ((len = getline(line, MAXLEN)) > 0) {
        if (lineptr[last] == NULL)
            if ((lineptr[last] = alloc(MAXLEN)) == NULL)
                error("tail: nem tud uj sort allokalni");
        nlines++;
        strcpy(lineptr[last], line);
        if (++last >= LINES)    /* atfordulas? */
            last = 0;
    }

    if (n > nlines)            /* tobb a kert sor a
                                   beolvasottnal? */
        n = nlines;
    first = last - n;
    if (first < 0)              /* atfordult a listan? */
        first += LINES;
    for (i = first; n-- > 0; i = (i + 1) % LINES)
        printf("%s", lineptr[i]);
}

error(s)
char *s;
{
    printf("%s\n", s);
    exit(1);
}

```

Az argc változó jelzi a parancssoron érkező argumentumok számát.

tail

esetén argc értéke 1.

tail -20

esetén argc értéke 2: a programnév, amit a -20 követ.

A program a bemenet utolsó  $n$  sorát nyomtatja ki. Ha `argc` értéke 1, akkor  $n$  az alapértelmezett `DEFLINES` értéket kapja. Ha `argc` értéke 2, akkor  $n$  a parancssorból kapja meg az értékét. Hibát jelent, ha `argc` értéke 2-nél nagyobb. A

```
while ((len = getline(line, MAXLEN)) > 0)
```

ciklus egyenként beolvassa a sorokat mindaddig, amíg a `getline` (1.14. Gyakorlat) meg nem találja a bemenet végét. Minden beolvasott sorhoz egy puffert allokalódik – feltéve, hogy az nem volt már korábban lefoglalva számára:

```
if (lineptr[last] == NULL)
    if ((lineptr[last] = alloc(MAXLEN)) == NULL)
        error("tail: nem tud uj sort allokalni");
```

A program a `len` számú karakter helyett `MAXLEN` karaktert allokal, mivel a puffert a program ismeretlen számú karakter számára a továbbiakban még használhatja (egészen `MAXLEN`-ig).

A `lineptr` tömb elemei karakterekre mutatnak: az eddig beolvasott utolsó `LINES` számú sorra. E tömb indexe a `last` változó. Kezdeti értéke 0 és minden beolvasott sor után eggyel növekszik:

```
if (++last >= LINES)
    last = 0;
```

Amikor a `last` egyenlővé válik `LINES`-szal, ismét nullázódik és a `lineptr` elemei és azok pufferei újra felhasználhatók.

A sorok teljes száma `nlines`. Az utolsó  $n$  sort kell kinyomtatni, így a kért sorok száma nem haladhatja meg a kapott sorok számát:

```
if (n > nlines)
    n = nlines;
```

Ha a sorok teljes száma meghaladta `LINES`-t, módosítani kell az utoljára megváltozott és az induló indexet:

```
if (first < 0)
    first += LINES;
```

Ezután a program kinyomtatja az utolsó  $n$  sort:

```
for (i = first; n-- > 0; i = (i + 1) % LINES)
    printf("%s", lineptr[i]);
```

Mivel  $i$  a `first` értékkel indul és  $n$  elemen halad át, megtörténhet, hogy közben újra előlről indult (át-



fordult). A moduló (maradék) operátor az  $i$  értékét 0 és  $LINES-1$  között tartja:

```
i = (i + 1) % LINES
```

Az `exit` standard könyvtári függvény (K-R 161. oldal) lezárja a programot hiba esetén. A hiba előfordulását `exit` az 1 érték visszaadásával jelzi.

### 5.11. Gyakorlat: (K-R 126. oldal)

Módosítsuk a `sort` programot oly módon, hogy kezelje `-rt`, amely ellentétes irányú (csökkenő) rendezést ír elő! `-r`-nek természetesen `-n`-nel is működnie kell!

```
#include <stdio.h>

#define NUMERIC 1    /* numerikus rendezes          */
#define DSORT  2    /* rendezes csokkeno sorrendben */
#define LINES 100    /* a rendezendo sorok max. szama */

static char option = 0; /* beallitott opciok bitjei */

main(argc, argv)      /* rendezi a bemeneti sorokat */
int argc;
char *argv[];
{
    char *s;
    char *lineptr[LINES]; /* mutatok a szovegsorokra */
    int nlines;           /* a beolvasott sorok szama */
    int strcmp(), numcmp(); /* összehasonlító függvények */

    int swap();           /* kicserelő függvény */

    while (--argc && (*++argv)[0] == '-')
        for (s = argv[0]+1; *s != '\0'; s++)
            switch (*s) {
                case 'n': /* numerikus rendezes */
                    option |= NUMERIC;
                    break;
                case 'r': /* rendezes csokk. sorrendben */
                    option |= DSORT;
                    break;
                default:
                    printf("sort: illegalis opcio %c\n", *s);
                    argc = 0;
                    break;
            }
    if (argc)
        printf("Hasznalat: sort -nr \n");
    else {
        if ((nlines = readlines(lineptr, LINES)) > 0) {
            if (option & NUMERIC) /* numerikus */
                sort(lineptr, nlines, numcmp, swap);
            else /* karakterlanc */
                sort(lineptr, nlines, strcmp, swap);
            writelines(lineptr, nlines, option & DSORT);
        }
    }
}
```

```

    } else
        printf("a bemenet túl nagy a rendezeshez\n");
    }
}

writelines(lineptr, nlines, order)      /* sorkezo */
char *lineptr[];
char order;
int nlines;
{
    int i;

    if (order)                          /* csokkeno rendben */
        for (i = nlines-1; i >= 0; i--)
            printf("%s\n", lineptr[i]);
    else                                /* novekvő rendben */
        for (i = 0; i < nlines; i++)
            printf("%s\n", lineptr[i]);
}

```

Az option nevű statikus karakterváltozó bitjei jelzik, hogy a híváskor mely opciókat kérték.

```

0. bit  = 0   karakterlánc rendezés
         = 1   numerikus rendezés           (--n)

1. bit  = 0   rendezés növekvő sorrendben
         = 1   rendezés csökkenő sorrendben (-r)

```

Ha híváskor volt előírt opció, a program a bitenkénti inkluzív VAGY operátor (!) segítségével állítja be az option változó megfelelő bitjét. Az

```
option |= DSORT;
```

utasítás egyenértékű az

```
option = option | 2;
```

utasítással. A decimális 2 szám ekvivalens a bináris 00000010 számmal. Mivel

```
1 VAGY valami = 1
```

ezért a fenti C utasítás az option karakterváltozó 1. bitjét 1-re állítja. (A bitek jobbról balra, 0, 1, 2,...).

Annak megállapítására, hogy valamelyik opció elő van-e írva, a bitenkénti ES (&) operátort használjuk.

Az

```
option & DSORT
```

kifejezés egyenlő 1-gyel, ha a -r opciót előírták és egyenlő 0-val, ha -r nincs előírva.

A writelines-t úgy módosítottuk, hogy az order (sorrend)



nevű haramadik argumentumot is fogadja. Az order változó értéke egyenlő a option & DSORT kifejezés eredményével, ami azt dönti el, hogy a rendezett listát növekvő vagy csökkenő sorrendben kell-e megjeleníteni.

Az strcmp, numcmp, swap, sort és readlines rutinok azonosak a sort programban használtakkal (K-R 124. oldal).

### 5.12. Gyakorlat: (K-R 126. oldal)

Illesszünk a sort programhoz egy -f opciót, amellyel egyesítjük a kis- és a nagybetűket úgy, hogy azokat a rendezés során nem különböztetjük meg: a kis- és a nagybetűs adatokat együtt rendezzük, tehát a és A együtt jelennek meg, nem választja el őket az egész ábécé.

```
#include <stdio.h>

#define NUMERIC 1 /* numerikus rendezes */
#define DSORT 2 /* rendezes csokkeno sorrendben */
#define FOLDING 4 /* kis es nagybetuk osszevonasa */
#define LINES 100 /* a rendezendo sorok max. szama */

static char option = 0; /* beallitott opciok bitjei */

main(argc, argv) /* rendezi a bemeneti sorokat */
int argc;
char *argv[];
{
    char *s;
    char *lineptr[LINES]; /* mutatok a szovegsorokra */
    int nlines; /* a beolvasott sorok szama */
    int charcmp(), numcmp(); /* összehasonlító függvények */
    int swap(); /* kicserelő függvény */

    while (--argc && (++argv)[0] == '-')
        for (s = argv[0]+1; *s != '\0'; s++)
            switch (*s) {
                case 'f': /* összevonas */
                    option |= FOLDING;
                    break;
                case 'n': /* numerikus rendezes */
                    option |= NUMERIC;
                    break;
                case 'r': /* rendezes csokk. rendben */
                    option |= DSORT;
                    break;
                default:
                    printf("sort: illegalis opcio %c\n", *s);
                    argc = 0;
                    break;
            }

    if (argc)
        printf("Hasznalat: sort -fnr \n");
    else {
```

```

    if ((nlines = readlines(lineptr, LINES)) > 0) {
        if (option & NUMERIC)          /* numerikus      */
            sort(lineptr, nlines, numcmp, swap);
        else                          /* karakterlánc */
            sort(lineptr, nlines, charcmp, swap);
        writelines(lineptr, nlines, option & DSORT);
    } else
        printf("a bemenet túl nagy a rendezéshez\n");
}
}

charcmp(s, t) /* a visszatérő érték <0, ha s<t,      */
char *s, *t; /* 0, ha s==t, >0, ha s>t            */
{
    int fold = (option & FOLDING) ? 1 : 0;
                                /* 1=összevonás */
    for( ; (fold?lower(*s):*s) == (fold?lower(*t):*t);
        s++,t++)
        if (*s == '\0')
            return(0);
    return((fold) ? (lower(*s) - lower(*t)) : (*s - *t));
}

```

A megoldás elve azonos az 5.11. Gyakorlatéval.

```

2. bit  = 0  nincs egyesítés
        = 1  egyesítés                                (-f)

```

Ha a híváskor kértük az egyesítést, az option második bitjét 1-re kell állítani:

```
option |= FOLDING;
```

Bináris jelölésmódban a decimális 4 (FOLDING) értéke 00000100 (a bitek számozása jobbról balra 0, 1, 2, ...).

A charcmp függvény a strcmp-hez (K-R 111. oldal) hasonlóan karakterláncokat hasonlít össze, de lehetővé teszi a kis- és a nagybetűs karakterek együttes rendezését.

Ha a híváskor előírtuk az együttes rendezést, akkor a charcmp rutin által összehasonlított két karaktert a program először kisbetűssé konvertálja. Ezután elvégzi az összehasonlítást és inkrementálja a mutatókat. Mindhárom lépést a for utasítás kezeli:

```
for( ; (fold?lower(*s):s) == (fold?lower(*t):t);
    s++,t++)
```

Azáltal, hogy a tároló tartalmának megváltoztatása nélkül minden karaktert kisbetűssé konvertálunk, egyesítjük a kis- és a nagybetűs karakterek kezelését. Ha tehát az egyberendezési opció elő volt írva, a visszaadott érték a két kisbetűs karakter távolsága, azaz (lower(\*s) - lower(\*t)). Megjegyzendő, hogy azonos eredményt kapnánk, ha minden karaktert nagybetűssé konvertálnánk.

A numcmp, swap, sort, readlines és writelines rutinok



azonosak az 5.11. Gyakorlatban használtakkal (K-R 124. oldal).

### 5.13. Gyakorlat: (K-R 126. oldal)

Illesszük a sort függvényhez a -d (szótári rendezés) opciót, amely csak betűket, számokat és szóközöket hasonlít össze! Ögyeljünk arra, hogy -f-fel együtt is működjön!

```
#include <stdio.h>
#include <ctype.h>

#define NUMERIC 1 /* numerikus rendezes */
#define DSORT 2 /* rendezes csokkeno sorrendben */
#define FOLDING 4 /* kis es nagybetuk osszevonasa */
#define DICTION 8 /* szotari sorrend */
#define LINES 100 /* a rendezendo sorok max. szama */

static char option = 0; /* beallitott opciok bitjei */

main(argc, argv) /* rendezi a bemeneti sorokat */
int argc;
char *argv[];
{
    char *s;
    char *lineptr[LINES]; /* mutatok a szovegsorokra */
    int nlines; /* a beolvasott sorok szama */
    int charcmp(), numcmp(); /* összehasonlító függvények */

    int swap(); /* kicserelő függvény */

    while (--argc && (*++argv)[0] == '-')
        for (s = argv[0]+1; *s != '\0'; s++)
            switch (*s) {
                case 'd': /* szotari rendezes */
                    option |= DICTION;
                    break;
                case 'f': /* osszevonas */
                    option |= FOLDING;
                    break;
                case 'n': /* numerikus rendezes */
                    option |= NUMERIC;
                    break;
                case 'r': /* rendezes csokk. rendben */
                    option |= DSORT;
                    break;
                default:
                    printf("sort: illegalis opcio %c\n", *s);
                    argc = 0;
                    break;
            }

    if (argc)
        printf("Hasznalat: sort -dfnr \n");
    else {
        if ((nlines = readlines(lineptr, LINES)) > 0) {
```

```

        if (option & NUMERIC)          /* numerikus      */
            sort(lineptr, nlines, numcmp, swap);
        else                            /* karakterlanc */
            sort(lineptr, nlines, charcmp, swap);
        writelines(lineptr, nlines, option & DSORT);
    } else
        printf("a bemenet tul nagy a rendezeshez\n");
}
}

charcmp(s, t)          /* a visszatero ertekek <0, ha s<t, */
char *s, *t;          /* 0, ha s==t, >0, ha s>t      */
{
    int fold = (option & FOLDING) ? 1 : 0;
    int dict = (option & DICTION) ? 1 : 0;

    do {
        if (dict) {          /* szotari sorrend */
            while (!isalnum(*s) && *s != ' ' && *s != '\0')
                s++;
            while (!isalnum(*t) && *t != ' ' && *t != '\0')
                t++;
        }
        a = (fold) ? lower(*s++) : *s++;
        b = (fold) ? lower(*t++) : *t++;
        if (a == b && a == '\0')
            return(0);
    } while (a == b);

    return(a - b);
}

```

A megoldás elve azonos az 5.11. és 5.12. Gyakorlatával.

```

3. bit  = 0  nincs szótári sorrend
        = 1  szótári sorrend                (-d)

```

Ha a híváskor kértük a szótári rendezést, az option harmadik bitjét 1-re kell állítani:

```
option |= DICTION;
```

Bináris jelölésmódban a decimális 8 (DICTION) értéke 00001000 (a bitek számozása jobbról balra 0, 1, 2, ...).

A charcmp függvényt (5.12. Gyakorlat) úgy módosítottuk, hogy mind az egyberendezési, mind pedig a szótári opciót kezelje.

A dict változó értéke 1, ha előírták a szótári opciót. A fold változó értéke 1, ha az együttes rendezési opció elő van írva.

Az a és b karakterváltozók ideiglenesen az s, illetve t által megcímzett értéket tartalmazzák. Ezáltal csökken a lower függvény szükséges hívásainak száma.

Ha a szótári rendezés elő van írva, akkor a



```
while (!isalnum(*s) && *s != " " && *s != '\0')
    s++;
```

ciklus minden s karakterláncbeli karaktert megvizsgál és átugorja azokat a karaktereket, amelyek nem betűt, számot vagy szóközt jelentenek. Az isalnum makró a szabványos C könyvtár részét képező ctype.h -ban van definiálva. Ez a fejléc (header) állomány definiálja a karakterek tulajdonságait vizsgáló gépfüggetlen ellenőrzéseket. Az isalnum azt vizsgálja, hogy egy karakter alfabetikus (a-z, A-Z), illetve numerikus-e (0-9). Ha \*s alfabetikus karakter vagy számjegy, akkor isalnum(\*s) nem nulla, egyébként pedig isalnum(\*s) nulla.

A következő while ciklus, azaz

```
while (!isalnum(*t) && *t != ' ' && *t != '\0')
    t++;
```

a t lánc minden karakterét megvizsgálja és átugorja azokat, amelyek nem betűk, számok vagy szóközök.

Ha a program s-ben és t-ben egyaránt betűt, számot vagy szóközt talál, akkor a két karaktert összehasonlítja.

A szótári opció miatt egy do-while ciklusra is szükség van, mivel az összehasonlítás (a==b) nem végezhető el a karakterek ellenőrzése előtt.

A numcmp, swap, sort, readlines és writelines rutinok azonosak az 5.11. Gyakorlatban használtakkal.

#### 5.14. Gyakorlat: (K-R 126. oldal)

Egészítsük ki a függvényt mezőkezelő szolgáltatással, amely lehetővé teszi, hogy sorokon belül kijelölt mezőkön is végezhető legyen rendezés, mégpedig minden mezőn egymástól független opciókészlet szerint! (E könyv angol kiadásának tárgymutatóját kulcsszavakra -df-fel, lapszámokra -n-nel rendezték.

```
#include <stdio.h>
#include <ctype.h>

#define NUMERIC 1 /* numerikus rendezes */
#define DSORT 2 /* rendezes csokkeno sorrendben */
#define FOLDING 4 /* kis- es nagybetuk osszevonasa */
#define DICTION 8 /* szotari sorrend */
#define LINES 1000 /* a rendezendo sorok max. szama */

static char option = 0; /* beallitott opciok bitjei */

main(argc, argv) /* rendezi a bemeneti sorokat */
int argc;
char *argv[];
{
```

```

char *lineptr[LINES];          /* szovegsor mutatok */
int i;
int nlines;                    /* beolvasott sorok szama */
int pos1 = 0;                  /* pos1-en kezdodo mezo */
int pos2 = 0;                  /* pos2 előtt végződő */
int charpcmp(), numpcmp();     /* összehasonlító
                                függvények */
int swap();                    /* felcserélő függvények */

readargs(argc, argv, &pos1, &pos2);
if ((nlines = readlines(lineptr, LINES)) > 0) {
    if (option & NUMERIC)      /* numerikus */
        sort(lineptr, nlines, numpcmp, swap, pos1, pos2);
    else                       /* karakterlánc */
        sort(lineptr, nlines, charpcmp, swap, pos1, pos2);
    writelines(lineptr, nlines, option & DSORT);
} else
    printf("a bemenet túl nagy a rendezéshez\n");
}

readargs(argc, argv, pos1, pos2) /* argumentumok
                                olvasása */

int argc;
char *argv[];
int *pos1, *pos2;
{
    char *s, t;

    while (--argc)
        if ((t = (*++argv)[0]) != '-' && t != '+')
            error("Hasznalat: sort -dfnr [+pos1][-pos2]");
        else if (t == '-' &&
                 (*(argv[0]+1) < '0' || *(argv[0]+1) > '9'))
            for (s = argv[0]+1; *s != '\0'; s++)
                switch (*s) {
                    case 'd': /* szotari rendezés */
                        option := DICTION;
                        break;
                    case 'f': /* összevonás */
                        option := FOLDING;
                        break;
                    case 'n': /* numerikus rendezés */
                        option := NUMERIC;
                        break;
                    case 'r': /* rendezés csokk. rendben */
                        option := DSORT;
                        break;
                    default:
                        printf("sort: illegalis opció %c\n", *s);
                        error("Hasznalat: sort -dfnr [+pos1][-pos2]");
                        break;
                }
        else if (t == '+') { /* pos1 */
            *pos1 = atoi(argv[0]+1);
            if (--argc)
                if ((*++argv)[0] == '-') { /* pos2 */
                    *pos2 = atoi(argv[0]+1);
                    if (--argc)

```



```

        error("Hasznalat: sort -dfnr [+pos1][-pos2]");
        else if (*pos1 >= *pos2 && *pos2 != 0)
            error("readargs: pos1 >= pos2");
        else
            break;
    } else
        error("Hasznalat: sort -dfnr [+pos1][-pos2]");
    else
        break;
} else {
    *pos2 = atoi(argv[0]+1);
    if (--argc)
        error("Hasznalat: sort -dfnr [+pos1][-pos2]");
    else if (*pos1 >= *pos2 && *pos2 != 0)
        error("readargs: pos1 >= pos2");
    else
        break;
}
}

sort(v, n, comp, exch, pos1, pos2) /* v[0] ... v[n] */
char *v[]; /* lancok rendezese */
int n, pos1, pos2;
int (*comp)(), (*exch)();
{
    int gap, i, j;

    for (gap = n/2; gap > 0; gap /=2)
        for (i = gap; i < n; i++)
            for (j = i-gap; j >= 0; j -= gap) {
                if ((*comp)(v[j], v[j+gap], pos1, pos2) <= 0)
                    break;
                (*exch)(v[j], v[j+gap]);
            }
}

charcmp(s, t, pos1, pos2) /* <0, ha s<t, 0, ha s==t, */
char s[], t[]; /* >0, ha s>t */
int pos1, pos2;
{
    char a, b;
    int i, j;
    int dict = (option & DICTION) ? 1 : 0;
    int fold = (option & FOLDING) ? 1 : 0;

    i = j = pos1;
    if (pos2 == 0) /* vegpozicio nincs megadva ? */
        if ((pos2 = strlen(s)) > strlen(t))
            pos2 = strlen(t); /* rovid lanc hossza */
    do {
        if (dict) { /* szotari sorrend */
            while (i < pos2 && !isalnum(s[i]) &&
                    s[i] != ' ' && s[i] != '\0')
                i++;
            while (j < pos2 && !isalnum(t[j]) &&
                    s[j] != ' ' && s[j] != '\0')
                j++;
        }
        if (i < pos2 && j < pos2) {

```

```

        a = (fold) ? lower(s[i++]) : s[i++];
        b = (fold) ? lower(t[j++]) : t[j++];
        if (a == b && a == '\0')
            return(0);
    }
    } while (a == b && i < pos2 && j < pos2);

    return(a - b);
}

#define MAXSTR 100

numpcmp(s1, s2, pos1, pos2) /* s1 es s2 numerikus */
char *s1, *s2; /* összehasonlítása */
int pos1, pos2;
{
    double atof(), v1, v2;
    char str[MAXSTR];

    substr(s1, pos1, pos2, str, MAXSTR);
    v1 = atof(str);
    substr(s2, pos1, pos2, str, MAXSTR);
    v2 = atof(str);
    if (v1 < v2)
        return(-1);
    else if (v1 > v2)
        return(1);
    else
        return(0);
}

substr(s, pos1, pos2, str, maxlen) /* s reszlanca */
char s[], str[];
int pos1, pos2, maxlen;
{
    int i, j, len;

    len = strlen(s);
    if (pos2 == 0) /* zero poz. nincs megadva ? */
        pos2 = len;
    if (len < pos1 || len < pos2)
        error("substr: a lanc tul rovid");
    if (pos2 - pos1 > maxlen)
        error("substr: a lanc tul hosszu");
    for (j = 0, i = pos1; i < pos2; i++, j++)
        str[j] = s[i];
    str[j] = '\0';
}

error(s) /* hibauzenet kiirasa es kilepes */
char *s;
{
    printf("%s\n", s);
    exit(1);
}

```



A megoldás elve azonos az 5.11., 5.12 és 5.13. Gyakorlatával.

A rendező parancs szintaxisa:

```
sort -dfr [+pos1] [-pos2]
```

Ha a sorokon belüli mezőkön történő rendezés elő van írva, akkor a rendezés pos1-en kezdődik és éppen pos2 előtt végződik. Ellenkező esetben pos1 és pos2 értéke 0, és a program az egész sort rendezi.

A readargs rutin beolvassa a parancssor argumentumait és beállítja a megfelelő változókat. A while utasítás minden ciklusában egy argumentum beolvasására kerül sor.

Az első if utasítás, vagyis

```
if ((t=(*++argv)[0] != '-' && t != '+'))
```

hamis, ha az argumentumokat helytelenül adták meg (nem előzi meg őket a mínuszjel (-) vagy a pluszjel (+)).

Az első else-if utasítás, azaz

```
else if (t == '-' &&  
         (*(argv[0]+1) < '0' || *(argv[0]+1) > '9'))
```

igaz, ha az argumentum mínuszjel, amit egy nemszámjegy követ. A switch utasítás ugyanúgy dolgozza fel az argumentumokat, mint az 5.11., 5.12. és 5.13. Gyakorlatban.

A következő else-if utasítás, azaz

```
else if (t == '+')
```

csak akkor igaz, ha a megadott argumentum az opcionális +pos1. A program kiszámítja +pos1 értékét és ellenőrzi, hogy -pos2 megjelenik-e a parancssorban:

```
if (--argc)  
    if ((*++argv)[0] == '-')
```

Ha megtalálta -pos2-t, akkor kiszámítja pos2 értékét. A program hibaüzenetet küld, ha pos1 nagyobb vagy egyenlő pos2-nél (feltéve, hogy pos2 meg van adva), továbbá akkor is, ha +pos1-et valami más követi, mint -pos2. Az error függvény kinyomtatja a hibaüzenetet és meghívja az exit függvényt (K-R 161. oldal), amely lezárja a programot.

Az utolsó else utasítás elvégzi a -pos2 feldolgozását abban az esetben, amikor +pos1 nincs a parancssoron.

A sort rutin módosul: pos1 és pos2 átadódik sort-nak, majd az összehasonlító rutinnak. A charpcmp a charcmp rutin módosított változata, amely a mezőket kezeli.

numpcmp ugyanúgy számokat hasonlít össze, mint numcmp, de felhasználja az új substr rutint, mivel az atof nem fogadja el a kezdőpontot és a hosszat, mint argumentumokat. Biztonságosabb, ha elkészítjük az új substr rutint, mintha a gyakran használt atof rutin felületét változtatnánk meg. A program elvégzi a részláncok rendezését, de ha kell, az egész sort is kicseréli.

A swap, readlines és writelines rutinok azonosak az 5.11. Gyakorlatban használtakkal.



## 6. FEJEZET

# Struktúrák

### 6.1. Gyakorlat: (K-R 135. oldal)

Módosítsuk a `getword` függvényt és mérjük meg a program sebességének változását!

```
#define DIG '0' /* számjegy */
#define LET 'a' /* betű */
int type[129] = {
    -1, 0, 1, 2, 3, 4, 5, 6, 7, 8,
    9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
    19, 20, 21, 22, 23, 24, 25, 26, 27, 28,
    29, 30, 31, ' ', '!', '"', '#', '$', '%', '&',
    '\'', '(', ')', '*', '+', '-', '.', ':', '/',
    DIG, DIG, DIG, DIG, DIG, DIG, DIG, DIG, DIG, DIG,
    ':', ';', '<', '=', '>', '?', 'U', LET, LET, LET,
    LET, LET, LET, LET, LET, LET, LET, LET, LET, LET,
    LET, LET, LET, LET, LET, LET, LET, LET, LET, LET,
    LET, LET, LET, 'I', '\\', 'J', '^', '_', ' ', LET,
    LET, LET, LET, LET, LET, LET, LET, LET, LET, LET,
    LET, LET, LET, LET, LET, LET, LET, LET, LET, LET,
    LET, LET, LET, LET, LET, '{', '}', '~', 127
};

getword(w, lim) /* a következő szó olvasása a
                  bemenetrol */
char *w;
int lim;
{
    int c, t;

    if (type[(c = *w++ = getch()) + 1] != LET) {
        *w = '\0';
        return(c);
    }
    while (--lim > 0) {
        t = type[(c = *w++ = getch()) + 1];
        if (t != LET && t != DIG) {
            ungetch(c);
            break;
        }
    }
    *(w-1) = '\0';
    return(LET);
}
```

A `type` függvényt az egész típusú, 129 elemű `type` tömb váltja fel. A tömb akkor használható az ASCII karakterkészlet számára, ha az EOF -1.

A `type` tömb indexe a karakter ASCII értéke plusz 1. Az 1

hozzáadása az EOF (-1) kezelése érdekében történik. Ha tehát a program az EOF-et kapja meg, akkor a `type[-1+1] = type[0]` elemhez fér hozzá, ha pedig az 'A'-t találja meg (ASCII értéke 65), akkor a `type[65+1] = type[66]` elemhez történik hozzáférés.

Ez a módosítás valamivel megnöveli a program sebességét.

## 6.2. Gyakorlat: (K-R 135. oldal)

Írjuk meg a `type` olyan változatát, amely független a karakterkészlettől!

```
#define DIGIT '0'
#define LETTER 'a'

type(c) /* visszaadja a karakter típusát */
int c; /* (karakterkészlettől független) */
{
    switch (c) {
        case 'A': case 'B': case 'C': case 'D': case 'E':
        case 'F': case 'G': case 'H': case 'I': case 'J':
        case 'K': case 'L': case 'M': case 'N': case 'O':
        case 'P': case 'Q': case 'R': case 'S': case 'T':
        case 'U': case 'V': case 'W': case 'X': case 'Y':
        case 'Z':
            return(LETTER);
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
            return(DIGIT);
        default:
            return(c);
    }
}
```

Az a megoldás, hogy minden nagybetűhöz, minden kisbetűhöz és minden számjegyre egy `case` utasítást rendelünk. Mivel nincs rá biztosíték, hogy az ábécé vagy a számkészlet mindenféle karakterkészleten belül folytonos tömböt alkot, az olyan tartományokat, mint amiket a `type`-ban is használtunk (K-R 135. oldal), azaz mint

```
(c >= 'a' & c <= 'z' || c >= 'A' && c <= 'Z')
```

és

```
(c >= '0' && c <= '9')
```

nem használhatunk.



### 6.3. Gyakorlat: (K-R 135. oldal)

Írjuk meg a kulcsszószámláló program olyan változatát, amely nem számlálja az idézőjelek közé zárt karakterláncokban előforduló kulcsszavakat!

```
#include <stdio.h>
#define DIGIT '0'
#define LETTER 'a'
#define MAXWORD 20

#define NKEYS (sizeof(keytab)/sizeof(struct key))

struct key{
    char *keyword;
    int keycount;
}keytab[]={ "break",0,
            "case", 0,
            /* .... */
            "while",0
        };

main() /* C kulcsszavak számlálása */
{
    int n, t;
    char word[MAXWORD];

    while ((t = getword(word, MAXWORD)) != EOF)
        if (t == LETTER)
            if ((n = binary(word, keytab, NKEYS)) >= 0)
                keytab[n].keycount++;

    for (n = 0; n < NKEYS; n++)
        if (keytab[n].keycount > 0)
            printf("%4d %s\n",
                keytab[n].keycount, keytab[n].keyword);
}

getword(w, lim) /* a következő szó olvasása a
                bemenetrol */
char *w;
int lim;
{
    int c, t;

    if ((t = type(c = *w++ = getch())) != LETTER) {
        if (t == '"') /* atlepi az idézőjelezett
                        karakterláncokat */
            while ((c = getch()) != '"' && c != EOF)
                if (c == '\\')
                    getch();
        *w = '\0';
        return(c);
    }
    while (--lim > 0) {
        t = type(c = *w++ = getch());
        if (t != LETTER && t != DIGIT) {
```

```

        ungetch(c);
        break;
    }
}
*(w-1) = '\0';
return(LETTER);
}

```

Ez a megoldás feltételezi, hogy az összes idézőjelnek megvan a párja. Ha ez nem így van, a megoldás nem használható.

Az egyetlen módosítás a getword rutinban történt. A main, binary és type rutinok, valamint a keytab struktúra változatlan. Ha a type idézőjelet ad vissza, akkor a while ciklus mindaddig újra és újra végrehajtódik, amíg a program egy másik idézőjelet vagy egy EOF-et nem talál. EOF előfordulása esetén a záró idézőjel hiányzik:

```

if (t == '"')
    while ((c = getch()) != '"' && c != EOF)
        if (c == '\\')
            getch();

```

Ha a program egy fordított törtvonalat (backslash-t) észlel a while cikluson belül, akkor a fordított törtvonalat követő karakter nem vevődik figyelembe. Ha idézőjelek között fordul elő egy fordított törtvonalat követő idézőjel, akkor a program azt nem értelmezi záró idézőjelnek. Például

```
printf("\\"hallo\\"");
```

hatására a

```
"hallo"
```

kimenet keletkezik.

#### 6.4. Gyakorlat: (K-R 141. oldal)

Írjunk olyan programot, amely beolvas egy C programot, és alfabetikus sorrendben kinyomtatja a változóneveknek azokat a csoportjait, amelyek első 7 karakterükben megegyeznek, azonban ezt követően valahol különböznek! Gyeljük arra, hogy a 7 paraméter legyen!

```

#include <stdio.h>
#define DIGIT '0'
#define LETTER 'a'
#define MAXWORD 20
#define YES 1
#define NO 0

struct tnode {
    char *word;
    /* alapcsomópont */
    /* a szovegre mutat */
};

```



```

    int match;          /* egyezest talalt */
    struct tnode *left;  /* bal oldali gyermek */
    struct tnode *right /* jobb oldali gyermek */
};

main(argc, argv)      /* abece sorrendben kiir minden */
int argc;              /* valtozonev csoportot, amely */
char *argv[];          /* num karakterben megegyezik */
{
    /* (7 = alapertelmezes) */
    struct tnode *root, *treex();
    char word[MAXWORD];
    int found;          /* =YES (1) ha egyezest talalt */
    int num;             /* ahany kar. kulonbozzon */
    int t;

    num = (--argc && (*++argv)[0] == '-') ?
        atoi(argv[0]+1) : 7;
    root = NULL;
    found = NO;
    while ((t = getword(word, MAXWORD)) != EOF) {
        if (t == LETTER && strlen(word) >= num)
            root = treex(root, word, num, &found);
        found = NO;
    }
    treexprint(root);
}

struct tnode *treex(p, w, num, found)
/* w elhelyezese p-nel vagy p alatt */
struct tnode *p;
char *w;
int num;
int *found;
{
    struct tnode *talloc();
    char *strsave();
    int cond;

    if (p == NULL) {
        p = talloc();
        p->word = strsave(w);
        p->match = *found;
        p->left = p->right = NULL;
    } else if ((cond = compare(w, p, num, found)) < 0)
        p->left = treex(p->left, w, num, found);
    else if (cond > 0)
        p->right = treex(p->right, w, num, found);
    return(p);
}

compare(s, p, num, found) /* szavak */
char *s;                  /* összehasonlítása és p->match */
struct tnode *p;          /* frissítése */
int num;
int *found;
{
    char *t = p->word;
    int i;

```

```

for (i = 0; *s == *t; i++, s++, t++)
    if (*s == '\0')
        return(0);
if (i >= num) { /* az első num kar.-ben azonos ? */
    *found = YES;
    p->match = YES;
}
return(*s - *t);
}

treexprint(p) /* a fa minden elemét kiírja, */
struct tnode *p; /* ha p->match = 1 */
{
    if (p != NULL) {
        treexprint(p->left);
        if (p->match)
            printf("%s\n", p->word);
        treexprint(p->right);
    }
}

type(c) /* visszaadja egy ASCII karakter típusát */
int c;
{
    if (c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z'
        || c == '_')
        return(LETTER);
    else if (c >= '0' && c <= '9')
        return(DIGIT);
    else
        return(c);
}

```

A num változó a változónevek azon karaktereinek a száma, amelyekre nézve a változónevek azonosak. Ha ezt a parancssoron nem adjuk meg, a program 7-re állítja be:

```

num = (--argc && (++argv)[0] == '-') ?
    atoi(argv[0]+1) : 7;

```

A found változó logikai típusú. found értéke YES (1), ha a szó num számú karakterében azonos a fa egyik szavával és NO (0), ha a program nem talált ilyen egyezést. found kezdeti értéke NO.

A szó akkor kerül be a fába, ha a getword rutin (K-R 134. oldal) által visszaadott érték LETTER valamint a szó hossza meghaladja num-ot vagy azzal egyenlő. A treex rutin, amely a tree (K-R 139. oldal) módosítása, elhelyezi a szót a fán. A treex rutin az strcmp helyett a compare rutint hívja.

A compare rutin a fán elhelyezendő szót valamelyik, már a fán lévő szóval hasonlítja össze. Ha az első num számú karakteren egyezést talál, akkor \*found és a fán talált szónak megfelelő egyezési tag (p->match) értéke YES (1) lesz:



```

if (i >= num) {
    *found = YES;
    p->match = YES;
}

```

A `treeexprint` rutin kinyomtatja azokat a fán lévő szavakat, amelyek legalább egy másik szóval azonosak.

```

if (p->match)
    printf("%s\n", p->word);

```

Annak érdekében, hogy a program az aláhúzás karaktert is betűnek tekintse, a `getword` által használt `type` rutint (K-R 135. oldal) módosítottuk. A C Programozási Nyelvben az aláhúzás (`_`) betűnek számít (K-R 45. oldal).

### 6.5. Gyakorlat: (K-R 141. oldal)

Írjunk elemi, kereszthivatkozó listát készítő (cross-referencing) programot, amely kinyomtatja egy dokumentumban előforduló szavak listáját, és minden szóra megadja azoknak a soroknak a sorszámát, ahol az illető szó előfordul!

```

#include <stdio.h>

#define LETTER 'a'
#define MAXWORD 20

struct linklist {
    /* a sorok sorszámainak kapcsolt listája */
    int lnum;
    struct linklist *ptr;
};

struct tnode {
    /* alapsomopont */
    char *word; /* a szövegre mutat */
    struct linklist *lines; /* sorok sorszámai */
    struct tnode *left; /* bal oldali leszarmazott */
    struct tnode *right; /* jobb oldali leszarmazott */
};

main() /* elemi kereszthivatkozó */
{
    struct tnode *root, *treeex();
    char word[MAXWORD];
    int t, linenum;

    linenum = 1;
    root = NULL;
    while ((t = getword(word, MAXWORD)) != EOF)
        if (t == LETTER)
            root = treeex(root, word, linenum);
        else if (t == '\n')
            linenum++;
    treeexprint(root);
}

```

```

struct tnode *treex(p, w, linenum)
    /* w elhelyezese p-nel vagy p alatt */
struct tnode *p;
char *w;
int linenum;
{
    struct tnode *talloc();
    struct linklist *lalloc();
    char *strsave();
    int cond;

    if (p == NULL) { /* uj szo érkezett */
        p = talloc();
        p->word = strsave(w);
        p->lines = lalloc();
        p->lines->lnum = linenum;
        p->lines->ptr = NULL;
        p->left = p->right = NULL;
    } else if ((cond = strcmp(w, p->word)) == 0)
        addln(p, linenum);
    else if (cond < 0)
        p->left = treex(p->left, w, linenum);
    else
        p->right = treex(p->right, w, linenum);
    return(p);
}

addln(p, linenum)
    /* sorszam hozzaadasa a kapcsolt listához */
struct tnode *p;
int linenum;
{
    struct linklist *temp, *lalloc();

    temp = p->lines;
    while (temp->ptr != NULL && temp->lnum != linenum)
        temp = temp->ptr;
    if (temp->lnum != linenum) {
        temp->ptr = lalloc();
        temp->ptr->lnum = linenum;
        temp->ptr->ptr = NULL;
    }
}

treexprint(p) /* a fa kinyomtatasa */
struct tnode *p;
{
    struct linklist *temp;

    if (p != NULL) {
        treexprint(p->left);
        printf("%10s: ", p->word);
        for (temp=p->lines; temp!=NULL; temp=temp->ptr)
            printf("%4d ", temp->lnum);
        printf("\n");
        treexprint(p->right);
    }
}

```



```

struct linklist *lalloc()
{
    char *alloc();

    return((struct linklist *)
           alloc(sizeof(struct linklist)));
}

struct tnode *talloc()
{
    char *alloc();

    return((struct tnode *)alloc(sizeof(struct tnode)));
}

```

A fa minden egyes szóra vonatkozóan egy szögpontot tartalmaz. Az egyes szögpontok tartalma a következő:

```

mutató a szó szövegére (*word)
mutató a számok egy kapcsolt listájára (*lines)
mutató a bal oldali leszármazott szögpontra (*left)
mutató a job boldali leszármazott szögpontra (*right)

```

A sorok számának kapcsolt listája egy linklist típusú struktúra. A lista minden eleme egy sornak a számát, valamint a kapcsolt lista következő elemét megcímző mutatót tartalmazza. Ha a listának nincs több eleme, a mutató értéke NULL.

A treex rutin a tree rutin (K-R 139. oldal) módosítása. treex elhelyezi a szót a fán annak a sornak a számával együtt, amelyen a szó a vonatkozó kapcsolt listán előfordul. Ha új szó érkezett, akkor a kapcsolt lista első eleméhez a sor száma rendelődik hozzá:

```
p->lines->lnum = linenum;
```

Ha olyan szó érkezik, amely már szerepel a fán, azaz

```
((cond = strcmp(w, p->word)) == 0)
```

akkor az addln rutin hozzáfűggeszti a sor számát a kapcsolt listához.

Az addln rutin végigmegy a kapcsolt listán, miközben keresi ugyanannak a sorszámnak vagy a NULL-nak az előfordulását:

```
if (temp->ptr != NULL && temp->lnum != linenum)
    temp = temp->ptr;
```

Ha a sornak a számát nem találta meg, a rutin a kapcsolt lista végére egy új elemet illeszt:

```
if (temp->lnum != linenum) {
    temp->ptr = lalloc();
    temp->ptr->lnum = linenum;
    temp->ptr->ptr = NULL;
}

```

Ha a sornak a számát sikerült megtalálni, akkor a program nem függeszti újra a kapcsolt listához a sor számát.

A `treeprint` rutin a `treeprint` (K-R 140. oldal) módosított változata. `treeprint` a fastruktúrát ábécé sorrendben kinyomtatja. A fastruktúrán előforduló minden egyes szó esetében a rutin kinyomtatja magát a szót és mindazoknak a soroknak a sorszámát, amelyekben a szó előfordult.

#### 6.6. Gyakorlat: (K-R 141. oldal)

Írjunk olyan programot, amely a bemeneten előforduló szavakat az előfordulás sorrendjébe rendezve nyomtatja ki! Minden szó elé írjuk oda az előfordulások számát!

```
#include <stdio.h>

#define LETTER 'a'
#define MAXWORD 20
#define NDISTINCT 1000

struct tnode {
    /* alapcsomópont */
    char *word; /* a szövegre mutat */
    int count; /* előfordulások száma */
    struct tnode *left; /* bal oldali leszármazott */
    struct tnode *right; /* jobb oldali leszármazott */
};

static struct tnode *list[NDISTINCT];
static int n = 0; /* a különböző szavak száma */

main() /* különböző szavak kiírása előfordulás
                                             szerint */
{
    struct tnode *root, *tree();
    char word[MAXWORD];
    int i, t;

    root = NULL;
    while ((t == getword(word, MAXWORD)) != EOF)
        if (t == LETTER)
            root = tree(root, word);
    treestore(root);
    sortl();
    for (i = 0; i < n; i++)
        printf("%2d:%20s\n", list[i]->count, list[i]->word);
}

treestore(p) /* fa tárolása a list[] tömbben */
struct tnode *p;
{
    if (p != NULL) {
        treestore(p->left);
        if (n < NDISTINCT)
            list[n++] = p;
    }
}
```



```

    treestore(p->right);
}
}

sortl() /* a lista rendezese */
{
    int gap, i, j;
    struct tnode *temp;
    for (gap = n/2; gap > 0; gap /= 2)
        for (i = gap; i < n; i++)
            for (j = i-gap; j >= 0; j += gap) {
                if ((list[j]->count) <= (list[j+gap]->count))
                    break;
                temp = list[j];
                list[j] = list[j+gap];
                list[j+gap] = temp;
            }
}

```

A megengedett egyedi szavak maximuma NDISTINCT. A tnode struktúra ugyanaz, mint ami a K-R könyv 138. oldalán található. \*list mutatók tömbje, minden mutató egy-egy tnode típusú struktúrára mutat. Az n változó az egyedi szavak darabszámát tartalmazó számláló. Mind az n, mind a \*list static-nak van deklarálva a rutinokat megelőzően, így ezek a main, treestore, sortl számára egyaránt ismertek, más file számára azonban nem.

Az inputon érkező minden szó a fán helyeződik el. A treestore rutin azután minden, a tnode-ot megcímző mutatót a list tömbben helyez el. list[n]n ugyanarra a struktúrára mutat, mint p. Ez a hozzárendelés mindaddig fennáll, amíg NDISTINCT-nél kevesebb a szavak száma.

A sortl rutin a shell rutin (K-R 69. oldal) módosítása. A sortl rutin a \*list tömböt list[j]->count szerint rendezi.

#### 6.7. Gyakorlat: (K-R 144. oldal)

Írjunk olyan rutint, amely a lookup és install által kezelt táblából töröl egy nevet és egy definíciót!

```

#include <stdio.h>

remove(s) /* az s nev es def. torlese hashtab-bol */
char *s;
{
    struct nlist *prev, *np;
    int h;

    prev = NULL;
    h = hash(s); /* az s lanc hash erteke */
    for (np = hashtab[h]; np != NULL; np = np->next) {
        if (strcmp(s, np->name) == 0)
            break;
    }
}

```

```

    prev = np;      /* feljegyzi az elozo bejegyzest */
}
if (np != NULL) {   /* megtalalta a nevet      */
    if (prev == NULL) /* elso a hash listaban ? */
        hashtab[h] = np->next;
    else             /* masutt a hash listaban */
        prev->next = np->next;
    free(np->name);
    free(np->def);
    free(np);        /* felszabaditja az allokalt */
}
/* strukturat */
}

```

A hashtab mutatótömb deklarációja a K-R. könyv 142. oldalán található. (A fordító megjegyzése.)

A remove rutin megkeresi az s karakterláncot a táblában. Ha a láncot megtalálta, kilép a ciklusból:

```

if (strcmp(s, np->name) == 0)
    break;

```

Ha az s karakterláncot nem találta meg, a for ciklus akkor ér véget, amikor az np mutató értéke NULL-lá válik.

Ha az np értéke nem NULL, akkor létezik a név, és a hozzátartozó definíciót el kell távolítani a táblából. A hashtab egyik bejegyzése egy kapcsolt lista kezdetére mutat. np a törlendő bejegyzésre, míg prev az np-t megelőző bejegyzésre mutat. Ha prev értéke NULL, akkor np az első bejegyzés a hashtab[h]-n kezdődő kapcsolt listában:

```

if (prev == NULL )
    hashtab[h] = np->next;
else
    prev->next = np->next;

```

Az np bejegyzés torlése után a név, a definíció és maga a struktúra számára lefoglalt hely felszabadul (lásd a free függvényt – K-R 183. oldal):

```

free(np->name);
free(np->def);
free(np);

```

#### 6.8. Gyakorlat: (K-R 144. oldal)

Az ebben a fejezetben közölt rutinokat alapul véve valósítsuk meg a #define előfeldolgozó egyszerű változatát, amely C programok számára használható! A getch és ungetch függvények hasznosak lehetnek.



```

#include <stdio.h>

#define DIGIT '0'
#define LETTER 'a'
#define MAXWORD 100

struct nlist { /* alaptabla bejegyzes */
    char *name;
    char *def;
    struct nlist *next; /* kov. bejegyzes a lancban */
};

main() /* a #define processzor egyszeru változata */
{
    int typ;
    char w[MAXWORD];
    struct nlist *p, *lookup();

    while ((typ = getword(w, MAXWORD)) != EOF)
        if (strcmp(w, "#") == 0) /* direktiva kezdete */
            getdef();
        else if (typ != LETTER) /* nem definialható */
            printf("%s", w);
        else if ((p = lookup(w)) == NULL)
            printf("%s", w); /* nincs definialva */
        else
            ungets(p->def); /* belepteti a definiciot */
}

getdef() /* definicio bevetele es elhelyezese */
{
    int i, typ;
    char def[MAXWORD], dir[MAXWORD], name[MAXWORD];

    skipblanks();
    if ((typ = getword(dir, MAXWORD)) != LETTER)
        error(dir[0],
            "getdef: # utan direktivat var");
    else if (strcmp(dir, "define") == 0) {
        skipblanks();
        if ((typ = getword(name, MAXWORD)) != LETTER)
            error(name[0],
                "getdef: nem-alfa - nevet var");
        else {
            skipblanks();
            for (i = 0; i < MAXWORD-1; i++)
                if ((def[i] = getch()) == EOF ||
                    def[i] == '\n')
                    break; /* definicio vege */
            def[i] = '\0';
            if (i <= 0) /* nem definicio? */
                error('\n', "getdef: hiányos define");
            else
                install(name, def); /* def.-t lerakja */
        }
    } else if (strcmp(dir, "undef") == 0) {
        skipblanks();
        if ((typ = getword(name, MAXWORD)) != LETTER)
            error(name[0], "getdef: nem-alfa az undef-ben");
    }
}

```

```

        else
            remove(name);
    } else
        error(dir[0],
            "getdef: # utan direktivat var");
}

error(c, s)                /* kiirja a hibauzenetet, */
int c;                     /* atlepi a sor maradekat */
char *s;
{
    printf("hiba: %s\n", s);
    while (c != EOF && c != '\n')
        c = getch();
}

skipblanks()               /* atlepi a szokozokat es tab-okat */
{
    int c;

    while ((c = getch()) == ' ' || c == '\t')
        ;
    ungetch(c);             /* visszahelyezi a nemszokoz kar.-t */
}

```

A main program az egyszerű előfeldolgozó program törzsét tartalmazza. A program feltételezi, hogy a direktívákat (define, undef) a # előzi meg, amelyet a getdef függvény kezel. Ha a getword-től kapott szó nem alfanumerikus, akkor nem szabad definícióban szerepelnie. A lehetséges definíciókat a program az alfanumerikus szavak között keresi. Ha egy definíció létezik, akkor azt az ungets függvény (4.4. Gyakorlat) megfordítva helyezi vissza a bemeneti szövegbe.

A direktívák kezelését a getdef függvény végzi:

```

#define    név        definíció
#undef     név

```

A program feltételezi, hogy a név alfanumerikus.

A definíción belül a

```

for (i = 0; i < MAXWORD-1; i++)
    if ((def[i] = getch()) == EOF ||
        def[i] == '\n')
        break;

```

ciklus addig olvassa be a definíciót, amíg a sor (vagy az állomány) végét meg nem találja. Ha a definíció létezik, akkor azt a program az install rutin (K-R 143. oldal) segítségével beépíti a táblába.

Az undef direktíva hatására a program eltüntet egy nevet a táblából (6.7. Gyakorlat).



## Bevitel és kivetel

## 7.1. Gyakorlat: (K-R 154. oldal)

Írjunk olyan programot, amely tetszőleges bemenetet képes ésszerű módon kinyomtatni! Minimális feladatnak a nemgrafikus karaktereket (a helyi szokásoknak megfelelően) oktálisban vagy hexadecimálisban nyomtassa ki, és hajtogassa össze a hosszú sorokat!

```
#include <stdio.h>
#include <ctype.h>

#define MAXLINE 100 /* max. karakterszam soronkent */
#define OCTLEN 6 /* oktális érték hossza */

main() /* tetszőleges bemenet értelmes kinyomtatása */
{
    int c, pos;

    pos = 0; /* sorbeli pozíció */
    while ((c = getchar()) != EOF)
        if (iscntrl(c) || c == ' ') { /* nemgrafikus vagy üres */
            pos = inc(pos, OCTLEN);
            printf(" \\%03o ", c);
            if (c == '\n') { /* ujsor karakter ? */
                putchar('\n');
                pos = 0;
            }
        } else { /* grafikus karakter */
            pos = inc(pos, 1);
            putchar(c);
        }
}

inc(pos, n) /* inkrementálja a kimeneti */
int pos, n; /* pozíciószámlálót */
{
    if (pos + n < MAXLINE)
        return(pos+n);
    else {
        putchar('\n');
        return(n);
    }
}
```

A kimeneti sor hossza MAXLINE. Az `iscntrl` makró definíciója a `ctype.h`-ban található, amely a szabványos C könyvtár része. Az `iscntrl` megkeresi a nemgrafikus karaktereket: a törlés (delete) karaktert (oktális 0177)

és a közönséges vezérlőkarakttereket (amelyek oktális 040-nél kisebbek). A program a szóközöket ugyancsak nemgrafikus karaktereknek tekinti. A nemgrafikus karaktereket a program oktális kódban, OCTLEN pozíción nyomtatja ki oly módon, hogy a karakter kódját egy szóköz és egy \ előzi meg. Az újsor karakter pos-t nullázza:

```
if (c == '\n') {
    putchar('\n');
    pos = 0;
}
```

Az inc függvény visszaadja az utolsó felhasznált pozíciót és ha a kimenet számára nem áll rendelkezésre n pozíció, akkor megtöri a sort.

## 7.2. Gyakorlat: (K-R 157. oldal)

Írjuk meg újra a 4. fejezetben látott kalkulátorprogramot úgy, hogy a bemenetre és a számkonverzióra a scanf és/vagy az sscanf függvényeket alkalmazzuk!

```
#include <stdio.h>
#include <ctype.h>
#define MAXSTR 80 /* maximalis lanchossz */
#define MAXUSE MAXSTR-2 /* tenyl. lanc: str+' '+'0' */
#define NUMBER '0' /* szam elofordulas jelzese */
#define TOOBIG '9' /* tul hosszú lanc jelzese */

getop(s, lim) /* veszi a kovetkezo operandust vagy */
char s[]; /* operatort */
int lim;
{
    int i = 1;
    int c = ' '; /* szokoz karakterrel kezd */
    char iostr[MAXSTR];
    static char lastchar[] = " ";

    sscanf(lastchar, "%1s", &c);
    if (c != ' ') /* mar elementett karaktert */
        lastchar[0] = ' '; /* torli az utoljara */
    else /* mentett karaktert */
        if (scanf("%1s", &c) == EOF)
            c = EOF; /* megtalalta a bemenet veget */
    if (c != '.' && !isdigit(c))
        return(c); /* operator vagy EOF */
    sprintf(iostr, "%c", c); /* karakter iostr-be */
    if (c != '.') /* a tort elotti szamjegyek */
        while (scanf("%c", &c) != 0 && i < MAXUSE)
            if (isdigit(c))
                sprintf(&iostr[i++], "%c", c);
            else
                break; /* nem szamjegy */
    if (c == '.') { /* tizedespont utani jegyek */
        if (i < MAXUSE)
            sprintf(&iostr[i++], "%c", c);
        while (scanf("%c", &c) != 0 && i < MAXUSE)
```



```

if (isdigit(c))
    sprintf(&iostr[i++], "%c", c);
else
    break; /* nem számjegy */
}
iostr[i++] = ' '; /* lanc-hatarolo sscanff-nek */
iostr[i] = '\0'; /* karakterlanc vege */
if (strlen(iostr)-1 < lim) { /* a szam rendben */
    sscanff(iostr, "%s", s);
    lastchar[0] = c; /* egy karakterrel tulment */
    return(NUMBER);
} else { /* a szam tul nagy */
    iostr[lim-1] = ' '; /* roviditi az akt. lancot */
    sscanff(iostr, "%s", s);
    while (scanf("%c", &c) != EOF && c != '\n');
    ; /* atlepi a sor maradekat */
    return(TOOBIG); /* tulcsordulas jelzes */
}
}

```

Egyedül a `getop` függvényt (K-R 89. oldal) módosítjuk.

A `getop` két hívása között ne feledkezzünk meg az olyan karakterekről, amelyek egy számot követnek. A `lastchar` olyan egyelemű statikus tömb, amely tárolja az utoljára beolvasott karaktert (`sscanf` karakterláncot fogad).

Az `scanf` és `sscanf` esetében a

```
"%1s"
```

vezérlés egy nemüres karaktert fogad.

Karakterbeolvasást az

```
sscanf(lastchar, "%1s", &c);
```

csak akkor végez, ha az előző híváskor utoljára előfordult karakter nemüres volt. Egyéb esetben `c` nem változik (továbbra is üres karakter). Ha az utolsó karakter üres volt, az

```
if (scanf("%1s", &c) == EOF;
```

utasítás beolvassa a bemenetről az első nemüres karaktert.

Az első nemüres karakter eldönti, hogy az aktuális bemenet operátor vagy operandus-e. Ha a karakter nem tizedespont, és nem is számjegy, akkor vagy operátor vagy pedig EOF. Ha viszont operandus, akkor az első karakter az `iostr` láncra helyeződik:

```
sprintf(iostr, "%c", c);
```

Ez a karakter egy operandus első karaktere volt. Akár tizedespont, akár számjegy lehetett. Ha nem tizedespont volt, a `getop` mindaddig egyesével olvassa be a karakter-

reket, amíg egy nemszámjegyre talál (ez a tizedespontot megelőző rész, ha éppen van ilyen). Ha getop tizedespontot talál (az operandus legelején, vagy a szám egész része után), akkor beolvassa a tizedespontot követő számjegyeket.

Az üres karakterek jelentik a scanf és sscanf számára a határolást.

Az

```
iostr[i++] = ' ';
```

utasítás az iostr-beli számjegyek láncának végét jelzi az sscanf számára. Az

```
iostr[i] = '\0';
```

utasítás pedig az strlen számára jelzi a karakterlánc végét.

Ha nem volt túlcsordulás (az iostr hossza kisebb, mint lim-é), akkor a getop a NUMBER-t és az iostr-ből átmásolt számjegyek láncát adja vissza, továbbá megjegyzi az utoljára beolvasott karaktert. Ha túlcsordulás volt, a getop a TOOBIG-et és az iostr-ből átmásolt lim-1 karaktert adja vissza, a bemeneti sor további részét pedig figyelmen kívül hagyja.

Az isdigit makró a ctype.h-ban van definiálva. Nem nullát ad vissza az igaz esetre, ha az argumentum egy számjegy (0-9) ASCII megfelelője, egyéb esetben pedig a hamis (0) értéket adja vissza.

### 7.3. Gyakorlat: (K-R 162. oldal)

Írjunk olyan programot, amelyik összehasonlítja két állományt, és kiírja az első olyan sort és karakterpozíciót, ahol az állományok eltérnek egymástól!

```
#include <stdio.h>
#define MAXLINE 100

main(argc, argv) /* két állomány összehasonlítása */
int argc; /* - fo program */
char *argv[];
{
    FILE *fp1, *fp2, *fopen();

    if (argc != 3) { /* az argumentumok száma hibás - ? */
        fprintf(stderr, "comp: ket allomanynevet adjon\n");
        exit(1);
    } else {
        if ((fp1 = fopen(*++argv, "r")) == NULL) {
            fprintf(stderr, "comp: %s nem nyithato meg\n", *argv);
            exit(1);
        } else if ((fp2 = fopen(*++argv, "r")) == NULL) {
```



```

fprintf(stderr, "comp: %s nem nyithato meg\n", *argv);
    exit(1);
} else {
    /* megtalalta es */
    /* megnyitotta az osszehasonlitando allomanyokat */
    filecomp(fp1, fp2);
    fclose(fp1);
    fclose(fp2);
    exit(0);
}
}

filecomp(fp1, fp2)          /* ket allomany soronkenti */
FILE *fp1, *fp2;           /* osszehasonlitasa */
{
    char line1[MAXLINE], line2[MAXLINE];
    char *lp1, *lp2;
    int pos;

    do {
        lp1 = fgets(line1, MAXLINE, fp1);
        lp2 = fgets(line2, MAXLINE, fp2);
        if (lp1 == line1 && lp2 == line2) {
            if ((pos = linecomp(line1, line2)) >= 0) {
                printf("az else elteres a %d pozicion", pos);
                printf(" in line\n%s\n", line1);
                lp1 = lp2 = NULL;
            }
        } else if (lp1 != line1 && lp2 == line2) {
            printf("az else elteres a 0 pozicion");
            printf(" in line\n%s\n", line2);
        } else if (lp1 == line1 && lp2 != line2) {
            printf("az else elteres a 0 pozicion");
            printf(" a\n%s\nsorban\n", line1);
        }
    } while (lp1 == line1 && lp2 == line2);
}

linecomp(s, t)              /* -1 -et ad vissza, ha s==t, */
char *s, *t;                /* pos-t egyebkent */
{
    int pos;
    for (pos = 0; *s == *t; s++, t++, pos++)
        if (*s == '\0')
            return(-1);
    return(pos);             /* a lancok egyeznek */
}                            /* a pozicio, ahol kulonboznak */

```

Az argumentumok száma szükségszerűen három: a program neve, és az azt követő két állománynév. A program megnyitja az állományokat, majd a filecomp soronként összehasonlítja őket.

A filecomp mindkét állományból beolvas egy sort. Az fgets függvény a beolvasott sort megcímező mutatót, illetve állomány vége esetén NULL értéket ad vissza. Ha lp1 és lp2 a nekik megfelelő állományokra mutatnak, egyik állománynak sincs vége, és a program összehason-

lítja a két sort. Ha a sorok nem azonosak, akkor a filecomp kiírja azt a pozíciót, ahol különböznek, valamint az első állomány aktuális sorát.

Ha lp1 vagy lp2 nem a neki megfelelő állományra mutat, valamelyik állomány véget ért (EOF) és a sor nem egyezik a 0 pozíción.

Ha sem lp1 sem lp2 nem a neki megfelelő állományra mutat, mindkét állomány véget ért (EOF) – és nem különböznek.

A linecomp függvény hasonló a strcmp függvényhez (K-R 110. oldal). A linecomp -1 értéket ad vissza, ha a sorok egyeznek. Egyéb esetben az első olyan pozíciót adja vissza, ahol a sorok nem egyeznek.

#### 7.4. Gyakorlat: (K-R 162. oldal)

Módosítsuk az 5. fejezet mintakereső programját oly módon, hogy a bemenetét argumentumokként megnevezett állományok halmazából vegye, vagy ha ilyenek nincsenek, akkor a szabványos bemenetről! Ki kell-e iratni az állomány nevét, ha a program egymással egyező sorokat talál?

```
#include <stdio.h>
#define MAXLINE 100

main(argc, argv)    /* megkeresi az 1. arg. szerinti */
int argc;           /* mintát */
char *argv[];
{
    char line[MAXLINE], pattern[MAXLINE], *s;
    long lineno = 0;
    int except = 0, number = 0;
    FILE *fopen(), *fp;

    while (--argc > 0 && (++argv)[0] == '-')
        for (s = argv[0]+1; *s != '\0'; s++)
            switch (*s) {
                case 'x':
                    except = 1;
                    break;
                case 'n':
                    number = 1;
                    break;
                default:
                    printf("find: illegalis opcio %c\n", *s);
                    argc = 0;
                    break;
            }
    if (argc >= 1)
        copy(*argv, pattern);
    else {
        printf("Hasznalat: find [-x][-n] minta [file . . .]\n");
        exit(1);
    }
}
```



```

    if (argc == 1) /* szabványos bemenet olvasása */
        fpat(stdin, "\0", pattern, except, number);
    else
        while (--argc > 0) /* a megadott áll. olvasása */
            if ((fp = fopen(++argv, "r")) == NULL) {
                fprintf(stderr, "find: %s nemzithato meg\n", *argv);
                exit(1);
            } else { /* megnyitotta a megadott áll.-t */
                fpat(fp, *argv, pattern, except, number);
                fclose(fp);
            }
        }

    fpat(fp, fname, pattern, except, number)
    FILE *fp; /* mintakereses */
    char *fname, *pattern;
    int except, number;
    {
        char line[MAXLINE];
        long lineno = 0;

        while (fgets(line, MAXLINE, fp) != NULL) {
            ++lineno;
            if ((index(line, pattern) >= 0) != except) {
                if (*fname != '\0') /* van allomanynev */
                    printf("%s - ", fname);
                if (number) /* kiirja a sor szamat */
                    printf("%ld: ", fname);
                printf("%s\n", line);
            }
        }
    }
}

```

A main program ugyanúgy dolgozza fel az opcionális argumentumokat, mint az 5. fejezetben (K-R 122. oldal). Ezt követően legalább még egy argumentumot vár – a mintát. Ha a mintát nem követik állománynevek, akkor a szabványos bemenetet használja. Ellenkező esetben megnyit egy megnevezett állományt. Mindkét esetben meghívja az fpat függvényt.

Az fpat függvény túlnyomórészt azonos az eredeti main programmal. Egymás után beolvassa a sorokat mindaddig, amíg az fgets (K-R 161. oldal) a NULL értéket nem adja vissza. Minden sorban keresi a megadott mintát. A lehetőségek a következők:

| (index(line, pattern) >= 0) | != except            | eredmény |
|-----------------------------|----------------------|----------|
| 0 (nem találta a mintát)    | != 0 (nincs megadva) | hamis    |
| 1 (megtalálta a mintát)     | != 0 (nincs megadva) | igaz     |
| 0 (nem találta a mintát)    | != 1 (meg van adva)  | igaz     |
| 1 (megtalálta a mintát)     | != 1 (meg van adva)  | hamis    |

Ha a fenti kifejezés értéke igaz, akkor fpat kinyomtatja az állomány nevét (kivéve, ha az a szabványos bemenet), a sor számát, ha az elő volt írva, valamint magát a sort.

## 7.5. Gyakorlat: (K-R 162. oldal)

Írjunk olyan programot, amely több állományt nyomtat ki, minden újabb állományt új oldalon, cím kiírásával kezd, és az oldalakat állományonként folyamatosan számozza!

```
#include <stdio.h>
#define MAXBOT 3 /* sorok max. szama a lap aljan */
#define MAXHDR 5 /* sorok max. szama a lap tetejen */
#define MAXLINE 100 /* egy sor maximalis merete */
#define MAXPAGE 66 /* sorok max. szama egy oldalon */

main(argc, argv) /* allomanyok kiiratas - */
int argc; /* mindegyik uj oldalra */
char *argv[];
{
    FILE *fopen(), *fp;

    if (argc == 1) /* nincs arg; szabv. bem. masolasa */
        fileprint(stdin, "");
    else
        while (--argc > 0)
            if ((fp = fopen(*++argv, "r")) == NULL) {
                fprintf(stderr,
                    "print: %s nem nyithato meg\n", *argv);
                exit(1);
            } else {
                fileprint(fp, *argv);
                fclose(fp);
            }
        exit(0);
}

fileprint(fp, fname) /* allomanynev kiiratas */
FILE *fp;
char *fname;
{
    long lineno = 1;
    long pageno = 1;
    long heading();
    char line[MAXLINE];

    fprintf(stdout, "\f"); /* all.-t kezd uj oldalon */
    while (fgets(line, MAXLINE, fp) != NULL) {
        if (lineno % MAXPAGE == 1)
            lineno = heading(fname, pageno++);
        fputs(line, stdout);
        ++lineno;
        if (lineno > MAXPAGE - MAXBOT)
            do {
                fprintf(stdout, "\n");
            } while (++lineno <= MAXPAGE);
    }
}
```



```

long heading(fname, pageno) /* elhelyezi a fejléctet */
char *fname;                /* es atlep MAXHDR sort */
long pageno;
{
    long ln = 3;

    fprintf(stdout, "\n\n");
    /* ket ures sor a lap tetejen */
    fprintf(stdout, "%s    page %d\n", fname, pageno);
    while (ln++ < MAXHDR)
        fprintf(stdout, "\n");
    return(ln);
}

```

A program hasonl6 a cat programhoz (K-R. 159. oldal).

A fileprint függvény két argumentumot fogad: egy megnyitott állomány mutatóját és az állomány nevének mutatóját (az állománynév üres karakterlánc, ha az állomány a szabványos bemenet). A fileprint beolvassa és kinyomtja a sorokat. A

\f

karakter a lapdobást (form feed) jelöli. Ennek hatására minden állomány új oldalon kezdődik.

A lineno változó számlálja a sorokat. A laphossz MAXPAGE. Ha a lineno MAXPAGE-dzsel vett maradéka 1, azaz

```

if (lineno % MAXPAGE == 1)
    lineno = heading(fname, pageno++);

```

akkor a fileprint új fejléctet helyez el és nullázza a lineno változót. A heading függvény kiírja az állomány nevét és az oldalszámot, majd elegendő számú új sor karaktert helyez el úgy, hogy a lap tetején MAXHDR számú sor kerüljön felhasználásra.

MAXBOT az üres sorok száma a lap alján.

## Csatlakozás a UNIX operációs rendszerhez

---

### 8.1. Gyakorlat: (K-R 169. oldal)

Írjuk át a 7. fejezetben látott cat programot úgy, hogy a read, write, open és close rutinokat használjuk azok szabványos könyvtárbeli megfelelői helyett! Vegezzünk kísérleteket a két változat egymáshoz viszonyított sebességének meghatározására!

```
#include <stdio.h>
#define BUFSIZE 512

main(argc, argv)      /* cat: állományok egyesítése */
int argc;              /* - olvasas/iras */
char *argv[];
{
    int fd;

    if (argc == 1)      /* nincs arg: szabványos */
        filecopy(0);    /* bemenetet masolja */
    else
        while (--argc > 0)
            if ((fd = open(*++argv, 0)) == -1)
                error("cat: %s nem nyithato meg", *argv);
            else {
                filecopy(fd);
                close(fd);
            }
    exit(0);
}

filecopy(fd)
int fd;
{
    int n;
    char buf[BUFSIZE];

    while ((n = read(fd, buf, BUFSIZE)) > 0)
        if (write(1, buf, n) != n)
            error("cat: iras hiba", NULL);
}

error(s1, s2)          /* hibauzenetet ír és kilep */
char *s1, *s2;
{
    printf(s1, s2);
    printf("\n");
    exit(1);
}
```



```
if ((fd = open(*++argv, 0)) == -1)
```

utasítás olvasásra nyit meg egy állományt és visszaad egy (egész típusú) állományleíró (file descriptor); hiba esetén -1 értéket ad vissza.

A filecopy függvény BUFSIZE darabszámú karaktert olvas be az fd állományleíró használva. A read függvény az éppen beolvasott karakterek bájt számát adja vissza. Mindaddig, amíg a bájtok száma nagyobb, mint 0, nem jelenik meg hiba; a 0 az állomány végét jelzi, -1 hibát jelez. A write függvény n bájtot ír, egyébként hiba fordult elő.

Ez a változat körülbelül kétszer olyan gyors, mint az eredeti, amely a K-R könyv 7. fejezetében található.

## 8.2. Gyakorlat: (K-R 170. oldal)

Világos, hogy az lseek a seek felhasználásával megírható és viszont. Írjuk meg mindkettőt a másik felhasználásával!

```
lseek(fd, offset, origin)          /* lseek a seek */
int fd, origin;                    /* felhasználásával */
long offset;
{
    seek(fd, (int) (offset / 512), origin + 3);
    seek(fd, (int) (offset % 512), 1);
}
```

lseek előállítása a seek segítségével:

Az offset (eltolás) nevű long típusú változó a blokkok egész típusú darabszámává konvertálódik, így

```
seek(fd, (int) (offset / 512), origin + 3);
```

a helyes blokkot határozza meg. Az origin (kiinduló pozíció) plusz a 3 állandó a lehetséges 0, 1, 2 értéket a 3, 4, illetve 5 értékké konvertálja. Az utóbbi értékek azt jelzik, hogy a blokkok számát 512-vel kell megszorozni, majd a kiinduló pozíció értelemszerűen 0, 1, illetve 2 szerint értelmeződik.

A következő utasítás, vagyis

```
seek(fd, (int) (offset % 512), 1);
```

a blokkon belül megkeresi a helyes pozíciót a pillanatnyi pozíciótól kiindulva. (offset % 512) jelenti a blokk egy tört részét, ha van ilyen.

```

seek(fd, offset, origin)          /* seek az lseek */
int fd, offset, origin;          /* felhasználásával */
{
    long l;
    int neworigin;

    if ((neworigin = origin % 3) == origin)
        l = (long) offset;        /* origin: 0, 1, 2 */
    else
        l = (long) (offset * 512); /* origin: 3, 4, 5 */
    lseek(fd, l, neworigin);
}

```

seek előállítása a lseek segítségével:

Az origin a 0,...,5 értékeket veheti fel. Ha az érték 3 és 5 között van, akkor ez azt jelenti, hogy az eltolást a blokkmérettel meg kell szorozni és hosszú egészzé kell konvertálni. Ha viszont az érték 0 és 2 között van, akkor az eltolást csak hosszú egészzé kell konvertálni.

### 8.3. Gyakorlat: (K-R 175. oldal)

Írjuk át az fopen és a \_fillbuf függvényeket úgy, hogy explicit bitműveletek helyett mezőket használunk.

```

#define "nstdio.h"                /* nemszabványos stdio.h */
#define PMODE 0644                /* I/O a tulajnak; 0 masnak */

FILE *fopen(name, mode) /* allomany megnyitasa, all. */
register char *name, *mode; /* mutatot visszaadja */
{
    register int fd;
    register FILE *fp;

    if (*mode != 'r' && *mode != 'w' && *mode != 'a') {
        fprintf(stderr, "%s nyitasi modja illegalis\n",
            mode, name);
        exit(1);
    }
    for (fp = _iob; fp < _iob + _NFILE; fp++)
        if (fp->_flag.is_read==0 && fp->_flag.is_write==0)
            break; /* szabad rest talalt */
    if (fp >= _iob + _NFILE) /* nincs szabad res */
        return(NULL);

    if (*mode == 'w') /* hozzaferes az allomanyhoz */
        fd = creat(name, PMODE);
    else if (*mode == 'a') {
        if ((fd = open(name, 1)) == -1)
            fd = creat(name, PMODE);
        lseek(fd, 0L, 2);
    } else
        fd = open(name, 0);
    if (fd == -1) /* meghiusult eleres mod */
        return(NULL);
}

```



```

fp->_fd = fd;
fp->_cnt = 0;
fp->_base = NULL;
fp->_flag.is_unbuf = 0;
fp->_flag.is_buf = 1;
fp->_flag.is_eof = 0;
fp->_flag.is_err = 0;
if (*mode == 'r') { /* olvasas */
    fp->_flag.is_read = 1;
    fp->_flag.is_write = 0;
} else { /* iras */
    fp->_flag.is_read = 0;
    fp->_flag.is_write = 1;
}
return(fp);
}

_fillbuf(fp) /* bemen. puffer allokalas es toltes */
register FILE *fp;
{
    static char smallbuf[_NFILE]; /* pufferetlen B/K */
    char *calloc();

    if (fp->_flag.is_read == 0 ||
        fp->_flag.is_eof == 1 ||
        fp->_flag.is_err == 1 )
        return(EOF);

    while (fp->_base == NULL) /* pufferterul. kereses */
        if (fp->_flag.is_unbuf) /* pufferetlen */
            fp->_base = &smallbuf[fp->_fd];
        else if ((fp->_base=calloc (_BUFSIZE, 1)) == NULL)
            fp->_flag.is_unbuf = 1;
        else
            fp->_flag.is_buf = 1;
    fp->_ptr = fp->_base;
    fp->_cnt = read(fp->_fd, fp->_ptr,
        fp->_flag.is_unbuf ? 1 : _BUFSIZE);
    if (--fp->_cnt < 0) {
        if (fp->_cnt == -1)
            fp->_flag.is_eof = 1;
        else
            fp->_flag.is_err = 1;
        fp->_cnt = 0;
        return(EOF);
    }
    return(*fp->_ptr++ & 0377); /* char pozitiv lesz */
}

```

Az `stdio.h` állományról készült másolat neve `nstdio.h` (nem szabványos `stdio.h`). A bitmező változtatások az `fopen` és `_fillbuf` függvényeknek csak ezen módosított változataira vonatkoznak.

Az `_iobuf` struktúrára vonatkozó `typedef` a K-R könyv 171. oldalán található. Az `_iobuf` egyik eleme

```
int _flag;
```

A `_flag` változó bitmezők felhasználásával definiálódik újra:

```
struct _flag_field {
    unsigned is_read   : 1;
    unsigned is_write  : 1;
    unsigned is_unbuf  : 1;
    unsigned is_buf    : 1;
    unsigned is_eof    : 1;
    unsigned is_err    : 1;
};
```

A `_flag_field` struktúra minta ezután az `_iobuf`-ban kerül felhasználásra:

```
typedef struct _iobuf {
    char *_ptr;
    int _cnt;
    char *_base;
    struct _flag_field _flag;
    int _fd;
} FILE;
```

Az `nstdio.h` fenti módosításával az `fopen` függvény (K-R 173. oldal) módosítása megtörtént.

Az

```
if ((fp->_flag & (_READ | _WRITE)) == 0)
    break;
```

utasításban a `_READ` és `_WRITE` értékek között VAGY művelet hajtódik végre:

|                     |                |                      |          |
|---------------------|----------------|----------------------|----------|
| <code>(_READ</code> | <code> </code> | <code>_WRITE)</code> |          |
| 01                  |                | 02                   | oktális  |
| 01                  |                | 10                   | bináris  |
| 11                  |                |                      | eredmény |

Eszerint az `if` utasítás igaz, ha a `_flag` mindkét alsó bitje nulla (sem írás, sem olvasás). Ez azt igazolja, hogy az `_iob` egyetlen eleme sincs írásra vagy olvasásra használva.

A bitmezőre a következő explicit vizsgálat végződik el:

```
if (fp->_flag.is_read == 0 && fp->_flag.is_write == 0)
    break;
```

A következő módosítás a bitek explicit beállítására szolgál:

```
fp->_flag.is_unbuf = 0;
fp->_flag.is_buf   = 1;
fp->_flag.is_eof   = 0;
fp->_flag.is_err   = 0;
```

A programban található másik módosítás, azaz



```
fp->_flag &= ~(_READ | _WRITE);
```

nullázza az olvasás- és az írásjelző biteket, mivel

|                  |    |          |
|------------------|----|----------|
| (_READ   _WRITE) |    |          |
| 01               | 02 | oktális  |
| 01               | 10 | bináris  |
| 11               |    | eredmény |
| ~(11)            |    |          |
| 1...100          |    |          |

Ezen érték és az fp->\_flag közötti bitenkénti ES művelet kinullázza a \_flag utolsó két bitjét (az olvasás és írás biteket). A többi bitet változatlanul hagyja.

A következő lépésben

```
fp->_flag |= (*mode == 'r') ? _READ : _WRITE;
```

a \_flag-et mode szerint állítja be. Ha a mode értéke 'r', akkor \_flag értéke \_READ lesz, egyébként pedig \_WRITE.

A bitmezőket alkalmazva, ha a mód 'r', akkor az is\_read bit 1-re állítódik. Ha nem, akkor az is\_write bit értéke lesz 1:

```
if (*mode == 'r') {
    fp->_flag.is_read = 1;
    fp->_flag.is_write = 0;
} else {
    fp->_flag.is_read = 0;
    fp->_flag.is_write = 1;
}
```

A \_fillbuf függvény hasonlóképpen módosul.

A \_fillbuf függvény az EOF értéket adja vissza a következő esetekben: ha az állomány nem volt olvasásra megnyitva, ha már előfordult egy állomány vége vagy ha a program már előzőleg hibát észlelt:

```
if ((fp->_flag & _READ) == 0 ||
    (fp->_flag & (_EOF | _ERR)) != 0)
    return(EOF);
```

E bitek vizsgálata bitmezőkkel:

```
if (fp->_flag.is_read == 0 ||
    fp->_flag.is_eof == 1 ||
    fp->_flag.is_err == 1)
    return(EOF);
```

Ezt követően az

```
if (fp->_flag & _UNBUF)
```

utasításból

```
if (fp->_flag.is_unbuf)
```

lesz és az

```
    fp->_flag != _UNBUF;
else
    fp->_flag != _BIGBUF;
```

programszegmensből

```
    fp->_flag.is_unbuf = 1;
else
    fp->_flag.is_buf = 1;
```

lesz. Az

```
fp->_flag & _UNBUF ? 1 : _BUFSIZE
```

feltételes kifejezés

```
fp->_flag.is_unbuf ? 1 : _BUFSIZE
```

-ra módosul. Végezetül

```
    fp->_flag != _EOF;
else
    fp->_flag != _ERR;
```

alakja

```
    fp->_flag.is_eof = 1;
else
    fp->_flag.is_err = 1;
```

lesz. Az ebben a gyakorlatban közölt programkód a UNIX 7. verziójában érvényes. Az ezt követő változatokban az `stdio.h` megváltozott.

#### B.4. Gyakorlat: (K-R 175. oldal)

Tervezzük és írjuk meg a `_flushbuf` és `fclose` rutinokat!

```
#include <stdio.h>
```

```
_flushbuf(x, fp)    /* kimeneti puffer allokalasa es */
char x;              /* uritese */
register FILE *fp;
{
    static char smallbuf[_NFILE]; /* pufferetlen B/K */
    char *calloc();
    int nc;              /* uritendo karakterek szama */

    if ((fp->_flag & _WRITE) == 0 ||
        (fp->_flag & _ERR) != 0)
        return(EOF);
    if (fp->_base == NULL) { /* nincs allokalt puffer */
```



```

    while (fp->_base == NULL)          /* helyet keres */
        if (fp->_flag & _UNBUF)        /* pufferetlen */
            fp->_base = &smallbuf[fp->_fd];
        else if ((fp->_base=calloc(_BUFSIZE,1)) == NULL)
            fp->_flag != _UNBUF;
        else
            fp->_flag != _BIGBUF;
    } else {                            /* mar van puffer */
        nc = fp->_ptr - fp->_base;
        if (write(fp->_fd, fp->_base, nc) != nc)
            return(EOF);                /* hiba: EOF-et ad vissza */
    }
    fp->_ptr = fp->_base;                /* puffer kezdete */
    *fp->_ptr++ = x;                    /* menti a pill. karaktert */
    fp->_cnt=((fp->_flag & _BIGBUF) ? _BUFSIZE : 1) - 1;
    return(x);
}

fclose(fp)                            /* allomany lezarasa */
register FILE *fp;
{
    int rc;                            /* visszateres kod */
    int nc;                            /* uritendo karakterek szama */
    rc = 0;                            /* 0-val ter vissza siker eseten */
    if (fp < _iob || fp >= _iob + _NFILE)
        return(EOF);                  /* hiba: ervenytelen mutato */
    if (fp->_flag & _WRITE)             /* van mit uriteni? */
        if ((nc = fp->_ptr - fp->_base) > 0)
            if (write(fp->_fd, fp->_base, nc) != nc)
                rc = EOF;              /* iras hiba */
    if (fp->_flag & _BIGBUF)             /* allokalta puffer */
        cfree(fp->_base);              /* felszabaditasa */
    fp->_ptr = NULL;
    fp->_cnt = 0;
    fp->_base = NULL;
    fp->_flag &= ~(_READ | _WRITE);
    return(rc);
}

```

Ha az állomány nem volt írásra megnyitva vagy hiba fordult elő,

\_flushbuf egy EOF-et ad vissza:

```

if ((fp->_flag & _WRITE) == 0 || (fp->_flag & _ERR) != 0)
    return(EOF);

```

\_flushbuf első hívásakor nincs puffer a kiviteli puffer és fp->\_base értéke NULL. Ekkor – ugyanúgy, mint a \_fillbuf-ban (K-R 174. oldal) – egy puffer foglalódik le. \_flushbuf további hívásai alkalmával a pufferbe mentett karakterek törölődnek:

```

nc = fp->_ptr - fp->_base;
if (write(fp->_fd, fp->_base, nc) != nc)
    return(EOF);

```

nc a pufferbeli karakterek száma. A \_flushbuf rutin nc

számú karakter írását végzi el, és hiba esetén EOF-et ad vissza.

`_flushbuf` argumentuma egy karakter és egy állománymutató. A hívás után a program vagy lefoglal, vagy pedig kiürít egy puffert. A következő lépés a karakterargumentum mentése a pufferbe:

```
*fp->_ptr++ = x;
```

A lehetséges karakterek száma a pufferben (`fp->_cnt`) ezután egygyel kevesebb, mint a puffer mérete, tekintettel az éppen elmentett karakterre.

Az `fclose` függvény garantálja, hogy `fp` állománymutató. Ha mégsem az, akkor az `fclose` az EOF-et adja vissza:

```
if (fp < _iob || fp > _iob + _NFILE)
    return(EOF);
```

Ha nincs hiba, akkor a visszaadott érték 0.

Ha az állomány írásra volt megnyitva, szükségessé válhat néhány karakter ürítése. Az `nc` változó tartalmazza a puffertelt karakterek darabszámát. Ha a `write` utasítás végrehajtása során hiba fordulna elő, akkor a visszaadott érték EOF. Ha nagy puffer volt lefoglalva, akkor a pufferterületet a program felszabadítja:

```
if (fp->_flag & _BIGBUF)
    cfree(fp->_base);
```

Az utolsó lépés úgy állítja helyre az `_iobuf` struktúra elemeit, hogy az `fopen` ne találjon értelmetlen értékeket valamelyik szabad helyen.

A gyakorlatban közölt programkód a UNIX 7. verziójában érvényes. Az ezt követő változatokban az `stdio.h` megváltozott. (Lásd az `stdio.h` állományt a K-R. könyv 171. oldalán – a fordító megjegyzése.)

### 8.5. Gyakorlat: (K-R 175. oldal)

A szabványos könyvtárban rendelkezésünkre áll az

```
fseek (fp, offset, origin)
```

függvény, amely azonos az `lseek` függvénnyel attól eltekintve, hogy `fp` állománymutató és nem állományleíró. Irjuk meg `fseek`-et! Gondoskodjunk arról, hogy az általunk írt `fseek` helyesen működjön együtt a könyvtár többi függvényei számára végzett pufferkezeléssel!

```
#include <stdio.h>
```

```
fseek(fp, offset, origin) /* seek állománymutatóval */
register FILE *fp;
```



```

long offset;
int origin;
{
    int nc;                /* uritendo karakterek szama */
    long rc = 0;           /* visszateres kod */
    long lseek();

    if (fp->_flag & _READ) {
        if (origin == 1)    /* pillanatnyi poz.-rol? */
            offset -= fp->_cnt;
            /* a pufferbeli karaktereket feljegyzi */
        rc = lseek(fp->_fd, offset, origin);
        fp->_cnt = 0;        /* nincs pufferelt karakter */
    } else if (fp->_flag & _WRITE) {
        if ((nc = fp->_ptr - fp->_base) > 0)
            if (write(fp->_fd, fp->_base, nc) != nc)
                rc = -1;
        if (rc != -1)        /* meg nem volt hiba? */
            rc = lseek(fp->_fd, offset, origin);
    }
    return(rc == -1 ? -1 : 0);
}

```

Az `rc` változó a visszaadott kódot tartalmazza. Hiba előfordulása esetén `-1`-re állítódik be.

`fseek`-en belül két helyzet fordulhat elő: az állomány olvasásra van megnyitva, illetve az állomány írásra van megnyitva.

Ha az állományt olvasásra nyitottuk meg, és az `origin` értéke `1`, akkor az eltolás (`offset`) a pillanatnyi pozícióhoz képest számítható (a többi eset: az `origin` értéke `0`, ekkor az eltolás az állomány elejéhez képest számítható, illetve az `origin` értéke `2`, amikor az eltolás az állomány végéhez képest számítható). Amikor az eltolás a pillanatnyi pozícióhoz képest számítható, az `fseek` számításba veszi azokat a karaktereket is, amelyek már a pufferben vannak:

```

if (origin == 1)
    offset -= fp->_cnt;

```

`fseek` ekkor meghívja `lseek`-et és eldobja a pufferben lévő karaktereket:

```

rc = lseek(fp->_fd, offset, origin);
fp->_cnt = 0;

```

Amikor az állomány írásra van megnyitva, `fseek` először a pufferelt karaktereket üríti ki, ha vannak ilyenek:

```

if ((nc = fp->_ptr - fp->_base) > 0)
    if (write(fp->_ptr, fp->_base, nc) != nc)
        rc = -1;

```

Ha nincs hiba, akkor a program meghívja `lseek`-et:

```
if (rc != -1)
    rc = lseek(fp->_fd, offset, origin);
```

Az `fseek` függvény a sikeres keresések után 0-t ad vissza.

#### B.6. Gyakorlat: (K-R 183. oldal)

A `calloc(n, size)` szabványos könyvtári függvény `n` darab `size` nagyságú objektumot megcímző mutatót ad vissza, a tárterület kezdeti nagysága nulla. Írjuk meg a `calloc` függvényt úgy, hogy az `alloc`-ot mintaként vagy hívott függvényként használjuk!

```
#include <stdio.h>
```

```
char *calloc(n, size)      /* n darab "size" meretu */
unsigned n, size;          /* elem allokalasa */
{
    unsigned i, nb;
    char *alloc(), *p, *q;

    nb = n * size;
    if ((p = q = alloc(nb)) != NULL)
        for (i = 0; i < nb; i++)
            *p++ = 0;
    return(q);
}
```

A `calloc` függvény `n` darab `size` méretű objektumot foglal le. A lefoglalandó bájtok teljes száma `nb`:

```
nb = n * size;
```

Az `alloc` olyan mutatót ad vissza, amely egy `nb` byte-ból álló tárterületre mutat. A `p` és `q` mutatókban megőrződik ennek a lefoglalt tárterületnek a kezdete. Ha a helyfoglalás sikeres volt, az `nb` darab lefoglalt bájt a 0 kezdeti értéket kapja:

```
for (i = 0; i < nb; i++)
    *p++ = 0;
```

A `calloc` a lefoglalt és inicializált tárterület kezdetét megcímző mutatót ad vissza.

#### B.7. Gyakorlat: (K-R 183. oldal)

Az `alloc` a méretre vonatkozó kérést anélkül fogadja el, hogy annak jogosságát ellenőrizné. A `free` azt hiszi, hogy az a blokk, amelynek felszabadítását tőle kéri, érvényes méretű mezőt tartalmaz. Javítsuk e programok minőségét azzal, hogy nagyobb gondot fordítunk a hibael-lenőrzésre!



```

#include <stdio.h>

#define MAXBYTES (unsigned) 10240

static unsigned maxalloc;
/* alokalt egysegek max. szama */
char *alloc(nbytes) /* altalanos celu tar alokator */
unsigned nbytes;
{
    HEADER *morecore();
    register HEADER *p, *q;
    register int nunits;

    if (nbytes > MAXBYTES) { /* MAXBYTES-nal nem tobb */
        fprintf(stderr,
            "alloc: tobb, mint %u byte nem allokalhato\n",
            MAXBYTES);
        return(NULL);
    }
    nunits = 1+(nbytes+sizeof(HEADER)-1)/sizeof(HEADER);

    /* . . . */ /* lasd K-R. 181. oldal */
}

#define NALLOC 128
/* egyszerre alokalt egysegek szama */
static HEADER *morecore(nu)
/* memoriat ker a rendszertol */
unsigned nu;
{
    char *sbrk();
    register char *cp;
    register HEADER *up;
    register int rnu;

    rnu = NALLOC * ((nu + NALLOC - 1) / NALLOC);
    cp = sbrk(rnu * sizeof(HEADER));
    if ((int)cp == -1) /* egyaltalan nincs hely */
        return(NULL);
    up = (HEADER *)cp;
    up->s.size = rnu;
    maxalloc = (up->s.size > maxalloc) ?
        up->s.size : maxalloc;
    free((char *) (up+1));
    return(allocp);
}

free(ap) /* az ap blokkot a szabad listara helyezi */
char *ap;
{
    register HEADER *p, *q;

    p = (HEADER *)ap - 1; /* mutasson a fejre */
    if (p->s.size == 0 || p->s.size > maxalloc) {
        fprintf(stderr,
            "free: %u egyseg nem szabadithato fel\n", p->s.size);
        return;
    }
}

```

```

    for (q=allocp; !(p > q && p < q->s.ptr); q=q->s.ptr)
        /* . . . */
    }
    /* lásd K-R. 183. oldal */

```

Az `alloc` függvény összehasonlítja a kért bájtok számát egy tetszőleges `MAXBYTES` állandóval. Válasszon olyan értéket `MAXBYTES` számára, amely a legjobban illeszkedik az Ön rendszeréhez!

Amikor a `morecore` új blokkot foglal le, a `maxalloc` nevű `static` változóban feljegyződik az eddig felhasznált legnagyobb blokk mérete. Ezáltal a `free` függvény visszaigazolhatja a `size` értékét: az nem lehet 0, és nem lehet nagyobb, mint a lefoglalt legnagyobb blokk.

### 8.8. Gyakorlat: (K-R 183. oldal)

Írjuk meg a `bfree(p, n)` rutint, amely az `alloc` és a `free` által kezelt szabad lista számára felszabadítja az `n` karakterből álló tetszőleges `p` blokkot! `bfree` használatával a felhasználó bármikor beiktathat a szabad listába egy statikus vagy külső tombot.

```

bfree(p, n)      /* n karakterbol allo tetszoleges p */
char *p;          /* blokk felszabaditasa */
unsigned n;
{
    register HEADER *hp;

    if (n < sizeof(HEADER))
        return(NULL); /* tul keves a hasznalhatosaghoz */
    hp = (HEADER *) p;
    hp->s.size = n / sizeof(HEADER);
    free((char *) (hp+1));
    return((int)hp->s.size);
}

```

A `bfree` rutin két argumentumot vár: egy `p` mutatót, valamint a karakterek darabszámát, `n`-et. A blokkot csak abban az esetben szabadítja fel, ha a mérete legalább `sizeof(HEADER)`, egyébként a `NULL` értéket adja vissza.

A program a `p` mutatót `HEADER` típusúvá módosítja a "cast" művelet segítségével és `hp`-hez rendeli hozzá:

```
hp = (HEADER *) p;
```

A blokk mérete `sizeof(HEADER)` egységekben:

```
hp->s.size = n / sizeof(HEADER);
```

Az utolsó lépés a `free` függvény hívása. Minthogy a `free` olyan mutatót fogad, amely közvetlenül a fejterület mögé



mutat, (hp+1)-et használunk. Végül (hp+1) "cast" révén (char \*) típusúvá alakul át.

A bfree rutin a NULL értéket adja vissza, ha a blokk túl kicsi, egyébként pedig a blokk sizeof(HEADER) egységekben értelmezett méretét adja vissza.

## A, Á

abs makró 50, 52, 53  
add program 77  
addln 107  
aláhúzás ( \_ ) 106  
alloc 74, 75, 86, 133, 134, 135  
allokált puffer 129  
allokált tárterület 133  
any 39, 40  
aposztróf 34  
argc 77, 80, 81, 83, 84, 85, 86, 88, 90, 92, 94, 95, 103, 104, 117, 118, 119, 121, 123  
argumentum, parancssor 77, 80, 83, 84, 85, 86, 88, 90, 92, 94, 95, 103, 104, 117, 118, 119, 121, 123  
argv 77, 80, 83, 84, 85, 86, 88, 90, 92, 94, 95, 103, 104, 117, 118, 119, 121, 123  
ASCII 14, 19, 22, 46, 100, 101  
atof 55, 56, 73, 74  
atoi 38, 71, 80, 81, 84, 85, 86, 103, 104  
állomány kinyomtatása 121  
állományleíró 124  
állományok összehasonlítása 117  
állomány vége (EOF) 13, 28

## B

backslash (fordított törtvonal) 103  
backspace karakter (\b) 15, 16  
balra mozgó operátor (<<) 41, 42, 43, 44  
bfree 135  
bináris karakterábrázolás 51  
bináris számrendszer 45, 89, 90, 91, 92, 93  
binary 102  
bitcount 45  
bitenkénti ÉS operátor (&) 40, 42, 45, 89, 90, 91, 127, 128  
bitenkénti kizáró VAGY operátor (^) 44, 67  
bitenkénti VAGY operátor ( | ) 42, 43, 88, 89, 90, 91, 93, 95, 127  
bitinvertálás 44  
bitművelet 125  
blokkstruktúra 67  
break utasítás 47  
buf 60, 61, 62

bufp 60, 61, 62  
\_BUFSIZE 126, 129, 130  
\b backspace karakter 15, 16

## C

calloc 126, 129, 130, 133  
case prefixum 47, 57, 58, 77, 88, 90, 92, 93, 94, 101, 119  
cat program 122, 123  
charcmp 91, 93  
charpcmp 96  
cím 64, 65  
címpoperátor (&) 64, 65  
C kulcsszavak 102  
clear 58, 59, 78, 79  
close 123  
compare 104, 105  
continue utasítás 54  
copy 23, 24, 53, 119  
creat 125  
ctype.h 94, 114, 117

## D

day\_of\_year 75, 76  
default prefixum 47, 58, 59, 78, 88, 90, 91, 92, 93, 95, 96, 101, 119  
#define 111  
detab 82, 83, 85  
detab program 29, 80, 83, 85  
do-while utasítás 50, 51, 52, 53, 82, 83

## E, É

echo\_quote 33, 34  
egész típusú logikai változó  
-, inword 18  
-, okloop 28  
egész típusú változó  
-, lower 11  
-, nb 13  
-, nl 13  
-, nt 13



-, step 11  
 -, upper 11  
 egyes komplement operátor (~) 40, 41, 42, 43, 44  
 else-if 18, 48  
 entab 80, 83, 84, 85  
 entab program 30, 80, 83, 84, 85  
 EOF 13, 28  
 erase 59  
 error 86, 97, 98, 112, 113, 123, 124  
 esettab 84, 85  
 ES logikai operátor (& &) 37  
 exit 86, 87, 97, 98, 117, 118, 121, 123  
 expand 47, 50

## F

fa 105, 108, 109  
 fclose 117, 118, 120, 121, 129  
 feltételes kifejezés (? :) 46  
 fgets 118, 119, 120, 121, 122  
 FILE 117, 118, 119, 121, 125, 129, 130, 131, 132  
 filecomp 117, 118  
 fileprint 122  
 \_fillbuf 125, 126  
 findnb 32  
 \_flag 125, 126  
 float 68  
 \_flushbuf 129  
 fopen 117, 119, 125, 126  
 fordított lengyel 77, 79  
 fordított törtvonal (backslash) 103  
 for utasítás 12, 37  
 főprogram 34  
 főrutin 34  
 fpat 120  
 fprintf 117, 118, 121, 122, 125, 133, 134  
 fputs 121, 122  
 free 110, 111, 134, 135  
 fseek 131  
 \f lapdobás karakter 122

## G

getbits 40  
 getch 61, 62, 68  
 getchar 34, 61, 62  
 getdef 112, 113  
 getfloat 68  
 getint 68

getline 24, 25, 26, 27, 28, 53, 70, 71, 74, 75, 86, 87  
 getop 73, 74, 116  
 gettype 78  
 getword 100, 102, 106, 109

## H

HEADER 133, 134, 135  
 heading 121, 122  
 hexadecimális 38, 51, 114  
 hibaellenőrzés 133  
 hisztogram 19, 20, 21, 22  
 hőmérséklet-átszámító program 10, 11, 12  
 htoi 38

## I

idézőjel 34  
 if-else utasítás 13, 14, 15, 16, 18, 19, 28, 47, 48  
 inc 115  
 in\_comment 33, 35  
 index 55, 72, 73, 120  
 indexelés 70, 75, 76  
 in\_quote 35  
 install 110, 111, 113  
 invert 44  
 invertálás 44  
 inword 18, 19  
 \_iob 125  
 isalnum makró 93, 94  
 iscntrl makró 114  
 isdigit makró 117  
 itoa 50, 51, 52, 53, 63, 71  
 itoar 63, 64  
 itob 51, 52  
 itoh 51, 52

## J

jobbra mozgó operátor (>>) 40, 41, 42, 43, 44

## K

kalkulátor 57, 58, 115  
 karakter  
 -, \b (visszaléptetés) 15, 16  
 -, EOF 13, 28  
 -, \f (lapdobás) 122

- , \n (újsor) 9, 12, 13, 47
- , nemgrafikus 114
- , nemszóköz 31
- , nemüres 116
- , \0 (nulla) 23, 24, 25, 26, 27
- , \t (tab) 12, 13, 15, 16, 17, 19, 20, 25, 29, 30, 47, 81, 82, 83
- karakterállandó 33
- karakterlánc 14, 33
- karakterlánc vége (\0) 26, 27
- karaktermutató 70, 71, 72
- karaktertömb 23, 24, 25, 26, 27
- keresztthivatkozó lista 106
- kettes komplementű aritmetika 45, 50
- kitevő 55, 56
- kizáró VAGY operátor (^) 44, 67
- komplement 41
- kulcsszámláló program 102

## L

- lalloc 107
- lapdobás karakter (\f) 122
- lebegőpontos szám 55, 68
- lefoglalt puffer 129
- lefoglalt tárterület 133
- leghosszabb sort kereső program 23
- legnagyobb negatív szám 50
- linecomp 118, 119
- linklist struktúra 109
- logikai ÉS operátor (& &) 37
- logikai negáció operátor (!) 37
- logikai VAGY operátor (|) 15, 17, 18, 19, 37
- logikai változó (egész típusú)
  - , inword 18
  - , okloop 28
- lookup 110, 111, 112
- lower 22, 46, 91, 92, 93, 96, 97
- lower (egész típusú változó) 11
- lseek 124, 125, 131, 132

## M

- main 103
- makró
  - , abs 50
  - , isalnum 94
  - , iscntrl 114
  - , isdigit 117
  - , strcpy 70
  - , swap 67

- mantissza 56, 57
- MAXCOL szimbolikus állandó 32
- megjegyzés 33, 35
- megjegyzés kezdete (/\*) 34
- megjegyzés vége (\*/) 34
- mezőkezelés 94
- mintakeresés 119
- moduló operátor (%) 57, 59
- month\_day 75, 76, 77
- morecore 133, 134
- mutató 28
- mutatótömb 109

## N

- nb egész típusú változó 13
- negálás 37
- negatív szám 85, 86
- nemgrafikus karakter 114
- nemszóköz 31
- nemüres karakter 116
- newpos 32
- \_NFILE 125, 129, 130
- nl egész típusú változó 13
- NONBLANK szimbolikus állandó 14
- nstdio.h 126, 127
- nulla karakter (\0) 23, 24, 25, 26, 27
- numcmp 90, 91, 94
- numpcmp 96, 97
- nt egész típusú változó 13
- \n újsor karakter 9, 12, 13, 47

## O

- okloop 28
- oktális 114
- open 123, 125
- operátor
  - , balra mozgató (<<) 41, 42, 43, 44
  - , bitenkénti ÉS (&) 40, 42, 45, 89, 90, 91, 127, 128
  - , bitenkénti kizáró VAGY (^) 44, 67
  - , bitenkénti VAGY (|) 42, 43, 88, 89, 90, 91, 93, 95, 127
  - , cím (&) 64, 65
  - , egyes komplement (~) 40, 41, 42, 43, 44
  - , jobbra mozgató (>>) 40, 41, 42, 43, 44
  - , kizáró VAGY (^) 44, 67
  - , logikai ÉS (& &) 37
  - , logikai negáció (!) 37
  - , logikai VAGY (|) 15, 17, 18, 19, 37
  - , moduló (%) 57, 59



–, unáris (&) 64  
–, unáris mínusz (–) 57, 58, 59  
\\0 nulla karakter 23, 24, 25, 26, 27

## P

parancssor, argumentum 77, 80, 83, 84, 85,  
86, 88, 90, 92, 94, 95, 103, 104, 117, 118,  
119, 121, 123  
pop 57, 58, 79  
prefixum  
–, case 45, 57, 58, 77, 88, 90, 92, 93, 94,  
101, 119  
–, default 47, 58, 59, 78, 88, 90, 91, 92, 93,  
95, 96, 101, 119  
printf 63  
printf 9, 10, 54  
printf 32  
program  
–, add 77  
–, cat 122, 123  
–, detab 29, 80, 83, 85  
–, entab 30, 80, 83, 84, 85  
–, hőmérséklet-átszámító 10, 11, 12  
–, kulcsszószámláló 102  
–, leghosszabb sort kereső 23  
–, sort 88, 90, 91, 92  
–, szavakat számláló 16, 17  
–, tail 85  
puffertúlsordulás 28  
push 79

## R

\_READ 127, 128, 130, 131, 132  
read 123  
readargs 98  
readlines 74, 75, 90, 91, 94  
rekurzió 28, 63, 64, 65, 66  
remove 26, 111  
reverse 26, 27, 28, 50, 51, 52, 53, 63, 64, 65,  
66, 71, 72  
reverser 65, 66  
rightrot 42  
rindex 55  
rotáció 42, 43

## S

sbrk 133, 134  
scanf 115  
search 36

seek 125  
settab 82, 83, 84  
shell 110  
sizeof 134, 135, 136  
skipblanks 113  
sorösszehajtás 31, 114  
sortl 109, 110  
sort program 88, 90, 91, 92  
sprintf 115  
squeeze 39  
sscanf 115  
statikus tömb 115, 135  
stdio.h 62, 126  
stderr 117, 118, 120, 121, 125  
stdin 120, 121  
stdout 121, 122  
step egész típusú változó 11  
strcat 69  
strcmp 53, 88, 91, 92, 105, 110, 111, 112  
strcpy 74, 75, 86, 87  
strcpy makró 70  
strlen 48, 60, 61, 103, 104  
strsave 104, 105, 107  
struktúra  
–, linklist 109  
–, tnode 110  
struktúramutató művelet (–>) 104, 105,  
107, 110, 111, 112  
struktúratömb 102  
struct\_flag\_field 126, 127  
struct\_iobuf 126, 127  
struct key 102  
struct linklist 106  
struct nlist 110, 111, 112  
struct tnode 103, 104, 106, 109, 110, 111,  
112  
substr 97, 98  
swap 90, 91, 94, 95  
swap makró 67  
switch utasítás 47, 57, 58, 77, 78, 88, 92, 95,  
96, 101, 119

## SZ

szabványos C könyvtár 114, 131, 132, 133  
számkonverzió 115  
szavakat számláló program 16, 17  
szimbolikus állandó 14, 17, 18, 31, 62, 88,  
90, 92, 95  
–, MAXCOL 32  
–, NONBLANK 14  
szintaktikai hiba 34, 35  
szó 18, 19, 20

szóelválasztó 18  
szóhosszúság 19, 20  
szóköz 12, 13, 14, 25, 29, 30, 80, 83, 84

## T

tab karakter (\t) 12, 13, 15, 16, 17, 19, 20,  
25, 29, 30, 47, 81, 82, 83  
tabpos 82, 83  
tabulátorstop 29, 30, 80, 84, 85  
tail program 85  
talloc 104, 105, 107  
tetszőleges bemenet kinyomtatása 114  
tnode struktúra 110  
többdimenziós tömb 75  
tömb  
– karakter 23, 24, 25, 26, 27  
– mutató 109  
–, statikus 115, 135  
– struktúra 102  
–, többdimenziós 75  
tömbindexelés 70  
tree 105, 109  
treeprint 108  
treestore 110  
treex 105, 109  
treexprint 106, 108  
tudományos jelölésmód 55  
type 100, 101, 103, 105  
typedef 126, 127  
\t tab karakter 12, 13, 15, 16, 17, 19, 20,  
25, 29, 30, 47, 81, 82, 83

## U

újrainicializálás 12  
újsor karakter (\n) 9, 10, 47  
unáris mínusz operátor (–) 57, 58, 59  
unáris operátor (&) 64  
\_UNBUF 129, 130  
#undef 113

ungetch 60, 61, 62, 68, 69  
ungets 60, 61  
uniq (UNIX szolgáltatás) 53  
upper egész típusú változó 11  
utasítás  
–, break 47  
–, continue 54  
–, do-while 50, 51, 52, 53, 82, 83  
–, for 12, 37  
–, if-else 13, 14, 15, 16, 18, 19, 28, 47, 48  
–, switch 47, 57, 58, 77, 78, 88, 92, 95, 96,  
101, 119  
–, while 12, 13

## Ü

üres karakter 116

## V

VAGY logikai operátor (||) 15, 17, 18, 19,  
37  
verem 57  
véletlenszerű hozzáférés 124, 125, 131, 132  
visszaléptetés karakter (\b) 15, 16

## W

while utasítás 12, 13  
write 123  
\_WRITE 127, 128, 130, 131, 132  
writelines 91, 94  
wordlength 41, 42

## Z

zárójelek 34, 35, 36



Kiadja a Műszaki Könyvkiadó  
Felelős kiadó: Szűcs Péter igazgató  
Felelős szerkesztő: Györke Tiborné okl. villamosmérnök  
Szerkesztő: Votisky Zsuzsa

88.0837/3 MSZH Nyomda

Műszaki vezető: Kőrösi Károly  
Műszaki szerkesztő: Molnár Zoltán  
A borítót tervezte: Molnár Zoltán  
A könyv formátuma: B5  
Ívterjedelme: 12,87 (A5)  
Azonossági szám: 80020  
MŰ: 4144-k-8890  
Készült az MSZ 5601 és 5602 szerint  
A kézirat lezárva: 1987. november

# SZÜV ORSZÁGOS ÜGYFÉLSZOLGÁLATI IRODAHÁLÓZAT

SZÜV megújulási folyamatának a nagyközönség által is érzékelt, látványos eleme a

## COMPUTER—M

elnevezésű hálózat kialakítása. A hálózat első tagja a budapesti **COMPUTER-M** Ügyfélszolgálati Iroda már 1985-ben megnyitotta üvegajtóit a nagyközönség előtt. 1986-ban további nyolc ilyen iroda nyílt az ország más nagyvárosaiban, s 1987-ben már minden megyében hasonló, többfunkciójú létesítmények segítik a számítástechnikai kultúra terjesztését.

A **COMPUTER-M** mindenütt olyan kompu-terem, ahol komputerek láthatók, bérelhetők, vásárolhatók, elsősorban „M” komputerek, mikroszámítógépek, tehát, s ennek révén a termék a számítástechnika-alkalmazás fórumainak is tekinthetők. Annál is inkább, mert a SZÜV e bemutatótermeket korántsem csupán a saját termékeinek bemutatására szánta, hanem bármely hozzáforduló partner számítástechnikában alkalmazható gyártmányát kívánja itt bemutatni, terjeszteni.

A **COMPUTER-M** persze minden városban, elsősorban komputer-market, egyfajta kiskereskedelmi egység tehát, ahol egyebek között hardverek és szoftverek adás-vétele folyik. A kiskereskedelmi funkció nem titkolt célja, hogy a SZÜV alaptevékenységét támogassa. Ezekben a boltokban ugyanis olyan számítástechnikai eszközöket árusítanak, amelyhez a SZÜV egyéb szolgáltatásokat — pl. szoftverfejlesztést, szervízt — is képes nyújtani.

## A COMPUTER-M hálózat címjegyzéke

1067 Bpest Lenin krt. 57—59.

Telefon: 224-838

Telex: 227610

5600 Békéscsaba, Kinizsi u. 4—6.

Telefon: 66/21-155

Telex: 083418

3300 Eger, Grónay u. 3.

Telefon: 36/10-522

Telex: 063405

7400 Kaposvár, Rákóczi tér 9—11.

Telefon: 82/13-026, 27

Telex: 013226

6000 Kecskemét, Horváth Döme u. 12.

Telefon: 76/29-162

Telex: 026494

4400 Nyíregyháza, Tanácsköztársaság tér 7.

Telefon: 42/13-788

Telex: 073334

6726 Szeged, Jobb fasor 6—10.

Telefon: 62/11-311, 566

Telex: 082311

7100 Szekszárd, Wesselényi u. 15—17.

Telefon: 74/16-822

Telex: 014363

4024 Debrecen, Vöröshadsereg u. 22.

Telefon: 52/17-497

Telex: 072387

8000 Székesfehérvár, Schönherz Z. u. 36—40.

Telefon: 22/16-330

Telex: 021230

9700 Szombathely, Tolbuczin u. 31.

Telefon: 94/14-534, 535

Telex: 037260

7633 Pécs, Szántó Kovács J. u. 3.

Telefon: 72/32-355

Telex: 012322

3100 Salgótarján, Rákóczi út 202.

Telefon: 32/11-477

Telex: 0229223

5002 Szolnok, József Attila u. 22—24.

Telefon: 56/17-200

Telex: 033202

8900 Zalaegerszeg, Mártírok útja 42—44.

Telefon: 92/14-390

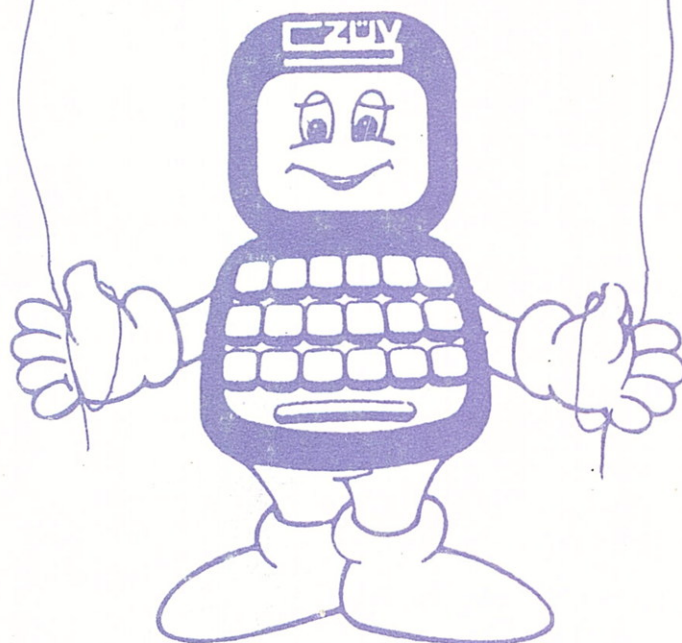
Telex: 033304





tanácsadás,  
oktatás,  
hardver-, szoftverbemutatók,  
programkészítés,  
helyi gépbérlet,  
szoftvertermékek,

papíralapú és  
mágneses adathordozók,  
hardvereszközök beszerzése,  
értékesítése  
számítástechnikai  
segédeszközök,  
szakirodalom,



COMPUTER-M